République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université M'hamed BOUGARA de BOUMERDES

Faculté des Sciences

Département d'Informatique

MEMOIRE DE MAGISTER

Spécialité : Systèmes informatiques et génie des logiciels

Option : Spécification de Logiciels et Traitement de l'Information

Ecole Doctorale

Présenté par :

LOUNAS Razika

Thème

Preuve en Coq de propriétés de programmes numériques partant du code en C

Devant le jury de soutenance composé de:

Mr. Mohammed AHMED NACER	Professeur	USTHB	Président
Mr. Mohammed MEZGHICHE	Professeur	UMBB	Rapporteur
Mme. Thouraya TEBIBEL	Maître de conférences	ESI	Examinateur
Mr. Ahmed AIT BOUZIAD	Maître de conférences	UMBB	Examinateur

Année Universitaire : 2008/2009

Remerciements

Mes remerciements les plus sincères vont en premier lieu à Monsieur Mohammed MEZGHICHE, Professeur à l'UMBB. Je le remercie pour m'avoir permis de travailler sur ce sujet et pour la confiance qu'il m'a accordée en me le proposant. Je le remercie également pour sa patience, ses encouragements et ses conseils précieux. Ce travail n'aurait jamais aboutit sans son précieux soutien et sa disponibilité. Je tiens également à le remercier en sa qualité de responsable de la Post Graduation pour les efforts qu'il fait pour nous, étudiants en PG.

Je tiens à remercier très chaleureusement Monsieur Marc DAUMAS Professeur à l'université de Perpignan (France) et Madame Sylvie BOLDO Chargée de recherches à l'INRIA (France) pour avoir d'abord rédigé les lignes qui ont constitué ce sujet passionnant puis pour leur précieuse aide.

Je remercie très chaleureusement les membres de mon jury de soutenance : Monsieur Mohammed AHMED NACER, Professeur à L'USTHB de m'avoir accordé l'honneur de présider mon jury de soutenance, Madame Thouraya TEBIBEL, maître de conférence à l' ESI et Monsieur Ahmed AIT BOUZIAD, maître de conférence à l'UMBB, d'avoir accepté d'examiner mon travail. J'espère que ce travail sera à la hauteur de leurs exigences scientifiques.

J'adresse mes sincères remerciements à Monsieur Kamel BADDARI Doyen de la faculté des science et à Monsieur Abdelkrim HARZELLAH, Chef du département d'informatique pour tous les efforts qu'ils font pour nous, pour le cadre serein, agréable et favorable au travail qu'on retrouve à la faculté des sciences et au département d'informatique.

Durant mes études à l'UMBB, j'ai eu l'occasion de croiser des personnes (personnels et enseignants) particulièrement au LIFAB et au département d'informatique. Je leur dis merci pour leur gentillesse et leur bonne humeur.

J'adresse un merci très sincère et amical à Malika, Samia et Zahira.

Un merci fraternel à mes sœurs et mes frères.

J'adresse un merci sincère, plein de reconnaissance et de gratitude aux deux personnes qui me sont le plus chères au monde : mes parents.

Résumé

L'utilisation des programmes informatiques dans des applications critiques nécessite l'utilisation des méthodes formelles basées sur la rigueur mathématique pour établir leur correction conformément à leurs spécifications.

La méthode formelle Why permet de générer, à partir d'un programme C spécifié avec Caduceus, un ensemble d'obligations de preuves qu'il faut prouver à l'aide d'un assistant de preuve pour établir la correction du programme.

Le calcul matriciel est intensivement utilisé dans les programmes scientifiques. Ceci a engendré le développement de plusieurs librairies dont BLAS (Basic Linear Algebra Subroutines), pour permettre une écriture rapide et efficace des programmes de calcul matriciel.

Dans notre travail, nous avons utilisé la méthode Why pour prouver deux programmes issus des BLAS : le produit matriciel et la résolution des systèmes.

Nous avons utilisé l'assistant de preuve Coq pour décharger les obligations de preuves. Pour mener les preuves, nous avons proposé une nouvelle définition du type matrice qui peut être utilisé pour prouver d'autres programmes.

Mots clés : Preuves des programmes, l'outil Caduceus, la méthode Why, l'assistant de preuve Coq, la librairie BLAS.

Abstract

The use of software in critical application requires the use of formal methods, based on mathematical accuracy, to prove the correctness of software regarding its specification.

The Why formal method generates, from a C program annotated using the Caduceus tool, a set of proof obligations. We must discharge these proof obligations within a proof assistant to establish the correction of the program.

Matrix calculus is intensively used in scientific computations. Consequently, many libraries such as BLAS (Basic Linear Algebra Subroutines) were developed to allow rapid and efficient writing of matrix programs.

In our work, we used the Why method to prove the correctness of two programs from the BLAS: matrix product and system resolution.

We used the Coq proof assistant to discharge the prove obligations. We defined also a new type matrix in Coq. This type could be used to prove other programs.

Keywords: Program proving, the Caduceus tool, the Why method, the Coq proof assistant, the BLAS library.

Table des Matières

Liste des figures

Liste des tableaux

Introduction Générale

Les programmes informatiques s'immiscent de plus en plus dans des applications dites critiques. Dans ce genre d'applications, une erreur logicielle peut avoir des conséquences dramatiques (pertes de vies humaines ou catastrophes économiques). C'est le cas des logiciels utilisés par exemple dans les transports ou dans la médecine.

Pour garantir la correction des programmes utilisés, les méthodes de tests sont insuffisantes. En effet, le domaine d'une application est souvent très grand, quand il n'est pas infini. Il en résulte qu'il est impossible d'effectuer des tests exhaustifs. Par conséquent, les tests peuvent seulement montrer la présence d'erreurs dans un programme mais ne garantissent jamais leurs absences.

La réponse aux exigences en matière de garantie de correction des programmes a été apportée par les méthodes formelles. Ces méthodes proposent d'abord un cadre formel pour décrire explicitement le comportement attendu d'un programme : sa spécification formelle. Ensuite, une démarche est proposée pour établir par un raisonnement mathématique la conformité d'un programme vis-à-vis sa spécification formelle. On a alors un programme certifié sans erreurs.

Les méthodes formelles de preuve des programmes sont scindées en deux parties : les méthodes de synthèse de programmes et les méthodes de vérification de programmes. La synthèse des programmes consiste à obtenir un programme certifié à partir d'une spécification formelle (par exemple, la méthode B). La vérification des programmes consiste à prendre des programmes déjà écrits et à démontrer leur correction.

La méthode formelle Why permet de prouver des programmes écrits en C. Cette méthode nous fournit un outils pour écrire des spécifications au cœur même du programmes : l'outils Caduceus. A partir du code annoté, cette méthode permet de générer un ensemble de formules logiques. L'utilisateur doit choisir un assistant de preuve, parmi ceux qui sont pris en charge par Why, pour démontrer ces formules et établir ainsi la correction du programme.

Notre travail s'inscrit dans la certification des programmes de calcul numériques. En effet, l'exigence du calcul scientifique en terme de rapidité et de performance des calculs a engendré le développement de logiciels et de bibliothèques scientifiques. Ces logiciels et bibliothèques sont intensivement utilisés dans la recherche et dans l'industrie. Plusieurs projets de recherches sont lancés pour certifier les programmes de calcul scientifiques. Citons le projet CerPAN qui vise à certifier des programmes d'analyse numériques.

Nous effectuons dans ce travail la preuve de deux programmes issus de la bibliothèque BLAS (Basic Linear Algebra Subroutines), en utilisant la méthode Why. Les BLAS constituent un standard incontournable dans le calcul matriciel. Les programmes que nous prouvons concernent le produit matriciel et la résolution des systèmes triangulaires.

Ce document est organisé comme ceci :

Le Chapitre 1 est une introduction aux preuves des programmes. Nous présentons dans un premier temps un aperçu sur les méthode formelles suivi d'une classifications des approches formelles des preuves des programmes. Dans un deuxième temps, nous nous intéressons aux systèmes de preuves avec un intérêt particulier pour les assistants de preuve.

Le chapitre 2 comporte deux sections. La première est consacrée à la présentation des techniques utilisées pour prouver des programmes impératifs. Il s'agit des logiques de Hoare et Dijkstra, des techniques de plongement des langages de programmation et des techniques de raisonnement sur la mémoire. La deuxième section est consacrée à un autre aspect des programmes : les calculs numériques qu'ils effectuent. Nous y présentons les principales approches d'interaction entre les preuves formelles et les propriétés numériques des programmes.

Le chapitre 3 concerne l'assistant de preuve Coq. C'est le système de preuve que nous utilisons pour notre travail. Nous présentons d'abord des aspects de son langage de spécification. Ensuite nous abordons un autre point concernant la manipulation des preuves. Nous terminons ce chapitre par un paragraphe concernant l'utilisation de Coq pour prouver des programmes.

Le chapitre 4 concerne les outils Caduceus et Why. Après une présentation générale de leur fonctionnement, nous présentons chaque outil en détail. Pour l'outil Caduceus, nous le présentons d'abord de point de vue langage de spécification concernant surtout les annotations des fonctions et des boucles. Nous le présentons ensuite de point de vue traduction. Il s'agit d'étudier les étapes qu'il effectue pour traduire un langage C en un langage écrit dans la syntaxe Du langage Why. Concernant l'outil Why, nous présentons d'abord ses caractéristiques générales puis, nous présentons ensuite en détail ses fondements théoriques. Il s'agit d'étudier comment cet outil génère les obligations de preuve. Ce chapitre se termine par un exemple détaillé d'utilisation de Caduceus et Why.

Le chapitre 5 est consacré à notre contribution. Celle-ci consiste à prouver deux programmes issus du domaine de l'algèbre linéaire et plus particulièrement de la librairie scientifique BLAS (Basic Linear Algebra Subroutines). Il s'agit des deux opération DGEMM_PLUS et DTRSM. Nous présentons d'abord les deux programmes et nous énonçons clairement l'objectif du travail et la démarche à suivre. Nous détaillons ensuite chaque étape de la solution.

Dans **La conclusion générale** nous récapitulons d'abord le travail réalisé. Nous présentons ensuite quelques perspectives que nous comptons explorer pour exploiter nos résultats, les enrichir et aborder d'autre aspects concernent les preuves des programmes.

CHAPITRE 1

Introduction
Aux preuves des programmes

Chapitre 1

Introduction aux preuves

Des programmes

1. Introduction

La propriété la plus importante qu'un programme doit vérifier est d'accomplir les tâches souhaitées par son utilisateur. Une grande partie du travail de programmation est en effet consacrée à la vérification de la correction des résultats d'une exécution. Couramment, pour se convaincre de la validité de son algorithme, le programmeur le teste sur des cas particuliers et le modifie si les résultats produits ne correspondent pas à son attente. Après une série de tests qui semblent suffisants, il estime que son programme est correct. Le temps passé à faire des tests est très souvent important par rapport au temps passé à écrire le programme [15]. De plus, le jeu de tests choisit n'est jamais exhaustif. Les tests permettent donc de montrer la présence d'erreurs dans un programme, mais ne peuvent jamais garantir leurs absences [50].

C'est dans cet esprit que des pionniers tels que C.A.R. HOARE [33] et E.W. Dijkstra [50] préconisèrent l'utilisation de méthodes rigoureuses pour établir la correction des programmes : ce sont les méthodes formelles.

2. Les méthodes formelles

L'expression "méthodes formelles" désigne l'utilisation des concepts et techniques issues des mathématiques ou de la logique formelle pour spécifier, développer et raisonner sur des systèmes informatiques et leurs propriétés ([51], [52]).

Comme la définition l'indique, les méthodes formelles peuvent être utilisées lors des différentes étapes du processus de développement d'un système. Une méthode formelle doit fournir [09] :

- Un langage formel permettant de décrire ou modéliser le système et ses propriétés;
- Une démarche articulée autours d'un ensemble d'outils pour raisonner sur des éléments de ce langage.

3. La spécification formelle

A la base des méthodes formelles, se trouve la notion de spécification formelle [51].Spécifier un programme consiste à donner sous une forme explicite sont comportement, c'est-à-dire décrire ce qu'il fait, mais pas comment il le fait. Il s'agit d'exprimer quelles sont ses données et quelles propriétés elles vérifient. Il s'agit aussi de spécifier quels sont les résultats (sorties) du programme et quelles propriétés ils doivent vérifier. On peut prendre un exemple avec la spécification d'un programme de division sur deux entiers naturels. Ce dernier a comme entrées les entiers *a* (dividende) et *b* (diviseur). Le diviseur doit vérifier la propriété *(b>0)*. Les sorties de ce programme sont les deux entiers *q* (quotient) et *r* (reste). Les entiers *q* et *r* doivent vérifier la propriété *(a = b*q+r et b>r)* [15].

Une spécification est dite formelle si [34] :

- Elle est écrite en suivant une syntaxe bien définie, comme celle des langages de programmation;
- La syntaxe est accompagnée d'une sémantique rigoureuse qui définit des modèles mathématiques représentant les réalisations acceptables de chaque spécification syntaxiquement correcte.
- La syntaxe et la sémantique sont donc accompagnées de règles de déduction qui permettent de démontrer des propriétés d'une spécification.

Concernant les propriétés, on en trouve deux types: les propriétés structurelles et les propriétés comportementales. Les propriétés structurelles concernant par exemple la composition des programmes et l'interconnexion entre les modules. Les propriétés comportementales se distinguent en deux types : Les propriétés fonctionnelles et les propriétés non fonctionnelles. Les propriétés fonctionnelles décrivent les liens entres entrées et les sorties du programme. Les propriétés non fonctionnelles décrivent en général toutes les autres contraintes par exemple les contraintes temporelles et l'aspect sécurité [51].

Les spécifications formelles permettent de définir le comportement d'un programme de façon rigoureuse, sans incohérence et sans ambiguïté. Plusieurs erreurs peuvent être évitées en écrivant la spécification formelle d'un programme.

Il est possible de spécifier formellement une application, sans pour autant utiliser de méthodes formelles pour la programmation et la vérification. En revanche, si l'on souhaite obtenir ou établir qu'un programme est certifié, ou garanti sans erreurs, il est nécessaire de l'avoir spécifié formellement.

Il existe plusieurs langages de spécification basés sur différents paradigmes de spécification.

3.1 Les paradigmes de spécification

3.1.1 Le paradigme fonctionnel [54] [56]

Le paradigme fonctionnel propose un cadre de spécification dans lequel la notion de fonction est centrale. Il se rapproche donc des langages de programmation fonctionnels comme Ocaml. Le système modélisé est exprimé par une série de définitions de fonctions, éventuellement récursives, sans effets de bord. Les fonctions sont représentées dans des formalismes dérivés du lambda-calcul qui est le fondement de tous les formalismes fonctionnels. Le lambda-calcul peut être enrichi avec des types pour les variables et les fonctions et offre un modèle se spécification très robuste. Le lambda-calcul typé est la base de certains systèmes comme L'assistant de preuve Coq [01].

3.1.2 Spécification à base d'états [54]

Ce paradigme consiste à spécifier le système par la donnée de préconditions et de postconditions pour les opérations du système et par la donnée d'invariants. La précondition d'une opération indique les conditions nécessaires pour que celle-ci soit appliquée. La postcondition d'une opération indique l'état à la fin de l'exécution de l'opération. Les invariants décrivent les contraintes sur les objets du système. Des langages tels que VDM [34], B [55] et Caduceus [06] reposent sur ce paradigme.

3.1.3 Le paradigme algébrique [34]

La spécification algébriques (ou par algèbre multi - sortées) est une collection d'ensembles de données, d'opérations et d'axiomes. Les noms des ensembles des valeurs sont appelés *sortes*. Par exemple, la sorte *nat* des nombres naturels. A chaque ensemble de données correspond un ensemble d'opérations. Chaque opération est typée par la donnée de son domaine et de son co-domaine. Les opérations sont spécifiées par un ensemble d'axiomes. Les axiomes sont soit des équations, soit des axiomes conditionnels. L'ensemble des sortes et des noms d'opérations d'une spécification algébrique est appelé *signature.* La méthode formelle Larch [14], par exemple, possède un langage de spécification algébrique.

3.1.4 Le paradigme état – transition [56]

Ce paradigme est représenté par les machines à états abstraits (ASM : Abstract State Machine). Les ASM forment un paradigme basé sur les notions d'états et de transitions. Un système y est modélisé par un ensemble d'états et par un ensemble de règles de transition. Cet ensemble décrit sous quelles conditions (appelées gardes) un ensemble de transformations ont lieu permettant la transition d'un état à un autre. Parmi les outils issus des ASM, citons le langage de programmation ASMGopher qui étend le langage de programmation fonctionnel Gopher.

3.1.5 Le paradigme logique [56]

Ce paradigme est centré sur les notions de prédicats, de règles et de déduction sur ces règles. Les données sont représentées à l'aide de types simples mais suffisamment expressifs (comme les listes). Une opération est décrite par ses propriétés comportementales, à l'image des axiomes des spécifications algébriques. La programmation logique, Prolog par exemple, considère les formules logiques comme des programmes et la construction de leurs preuves

comme l'exécution de ces programmes. Une approche similaire au model logique est donnée par les systèmes de réécriture tels que ELAN et SPIKE où les axiomes et règles de la logique sont remplacés par des équations déterminant le comportement calculatoire des fonctions.

4. Les preuves des programmes

Prouver un programme consiste à montrer par un raisonnement mathématique que le comportement induit par sa spécification est cohérent. L'intérêt des preuves des programmes est qu'elles nous permettent d'être sûrs à 100% qu'un programme est correct. On parle alors de programmes certifiés sans erreurs. De plus, un programme prouvé est un programme documenté. L'utilisateur sait exactement comment l'utiliser et ce qu'il fait et pourquoi il est correct.

On distingue deux principales approches permettant d'avoir des programmes certifiés ([13], [53]) : la synthèse des programmes et la vérification des programmes.

4.1 La synthèse des programmes

Nous retrouvons dans cette approches deux techniques : l'extraction et le raffinement.

4.1.1 L'extraction

L'extraction ([13], [03]) est adaptée à la certification de programmes fonctionnels. Elle est utilisée par les systèmes basés sur la logique constructive et l'isomorphisme de Curry – Howard. Dans la logique constructive, la preuve de la spécification d'une fonction donne une méthode pour construire celle-ci. L'isomorphisme de Curry – Howard permet de traiter les preuves comme des objets de calcul et d'en extraire ainsi une partie algorithmique.

L'idée est que tout énoncé logique est aussi une spécification. Une preuve de la spécification contient une information logique (qui nous assure que la démonstration est correcte) et une information dite calculatoire qui est extraite en tant que programme.

Prenons l'exemple [15] de la division de deux entiers naturels précédant et donnons la preuve mathématique de la spécification $\forall a, b\ (b>0) \rightarrow \exists\ q, r\ (a=b*q+r\ /\backslash b>r)$.Il s'agit d'une preuve par récurrence sur *a* :

Si *a = 0* ***alors*** *q = r=0* et on a bien : *0=b*0+0 et b>0.*
Sinon, on suppose qu'on connaît *q* et *r* pour le prédécesseur de *a*, c'est-à-dire : *(a-1)=b*q+r et b>r* et on essaie de *construire* q_1 et r_1 pour *a*. Il faut distinguer deux cas :

- ***Cas1 :*** $b \leq (r+1)$ et on a bien le quotient q_1 est *q+1*, le reste est *0* et *a =b*(q+1) +r et b>0*. Ceci provient du fait que *a-1 =b*q+r* et que *b>r* et $b \leq (r+1)$ donc *b=r+1*.
- ***Cas2 :*** *b>(r+1)* et on a le quotient résultat est *q,* le reste est *(r+1)* et on bien la propriété *a =b*q+r+1* et *b>(r+1).*

La preuve contient une méthode de construction explicite de *q* et *r (partie calculatoire),* mais elle prend aussi en argument une preuve que *(b>0)* et rend une preuve de *(a=b*q+r /\b>r)* pour chacun des cas possible *(partie logique).* C'est la partie calculatoire qui est donc extraite en tant que programme. On retrouve cette approche dans le système Coq qui propose une extraction vers les langages fonctionnels Haskell et Ocaml.

L'extraction pose le problème de la définition du contenu algorithmique : si on veut extraire une fonction faisant le tri d'une liste par exemple, on est obligé d'écrire une preuve correspondant à un algorithme choisi pour le tri.

4.1.2 Le raffinement

Le raffinement ([13], [55]) est issu des travaux d'axiomatisation des langages de programmation de MORGAN et DIJKSTRA [13]. Contrairement à la méthode précédente, celle-ci est beaucoup plus adaptée au traitement des programmes impératifs. Cette approche permet la synthèse pas à pas d'une description implantable à partir d'une spécification.

Chaque degré de précision supplémentaire représente un choix d'implémentation, par exemple, le choix de l'algorithme implémentant une fonction, ou encore le choix de la représentation concrète d'un type de donnée.

Chaque étape du raffinement engendre des formules logiques qu'il faut démontrer. Cela permet de garantir que l'étape *n+1* satisfait bien ce qui est requis à l'étape *n*. Ainsi, on est sur que le programme final jouisse des propriétés de la spécification originale. Le principal système faisant appel au raffinement est la méthode B. La figure (1.1.a) montre l'exemple d'une machine abstraite B .Son raffinement est donné par la figure (1.1.b)

```
MACHINE SquareRoot

OPERATIONS

sqrt ← SquareRoot (xx) =

   PRE xx ∈ N
  THEN
      ANY yy
      WHERE yy ∈ N /\ square (yy) ≤ xx /\ xx
< (square (yy+1)
      THEN sqrt :=yy
      END
 END
DEFINITIONS
square (x) = x*x
END
```

Figure 1.1.a La machine abstraite SquareRoot

```
REFINEMENT SquareRootR
REFINES SquareRoot
OPERATIONS

sqrt ← SquareRoot (xx) =
   ANY yy, zz
 WHERE yy ∈ N /\ zz ∈ N /\
          sqinv (xx, yy, zz) /\
           zz = yy +1
  THEN
   sqrt :=yy
END
DEFINITIONS
square (x) = x *x;
sqinv (x, y, z) = y <z /\ square (y) ≤ x /\
x<square (z)
END
```

Figure 1.1.b Raffinement de SquareRoot

La machine *SquareRoot* permet de spécifier la fonction qui calcule la racine carrée d'un nombre naturel. Elle définit une macro *square* et une opération appelée *SquareRoot*. Cette opération prend un argument naturel *xx* et retourne un nombre naturel *sqrt*. Le résultat de l'opération est défini par une variable *yy* qui satisfait un prédicat qui dit que *yy* est le plus grand nombre naturel dont le carré n'excède pas *xx*. L'écriture *sqrt :=yy* veut dire que la valeur de *sqrt* devient *yy*. Le raffinement de cette machine suggère un algorithme permettant de calculer la racine carrée de sorte qu'elle satisfasse toujours le prédicat.

Le raffinement pose plusieurs problèmes, citons par exemple l'optimisation et la maintenance. Pour la maintenance, toute modification au niveau du code obtenu, nécessite de revoir toute la chaîne de raffinement

4.2 La vérification des programmes

L'approche vérification consiste à prendre un programme déjà écrit, et à en vérifier sa spécification formelle [13]. Pour vérifier un programme, on procède en montrant que chacune des parties du programme accomplit son calcul, et que la combinaison des différentes parties du programme vérifie aussi les propriétés attendues.

Contrairement aux deux méthodes précédentes (qui sont des approches *« programmer en prouvant »*), celle-ci peut donc s'exercer *a posteriori* sur un programme existant. Elle a donc l'avantage de dissocier la partie programmation de la partie preuve. Les problèmes du contenu algorithmique et de l'optimisation ne sont pas posés non plus : le programme nous est donné. Nous distinguons deux techniques dans cette approche : L'annotation des programmes et la génération de spécification.

4.2.1 L'annotation des programmes

Cette technique consiste à insérer dans le code source des énoncés logiques décrivant son comportement : les annotations. Il s'agit du modèle de la logique de Hoare [33]. L'annotation d'une fonction ou d'un bout de code indique ses invariants en cours d'exécution, sa précondition et sa postcondition. La précondition représente les conditions préalables à l'exécution du code. La postcondition représente l'état à la fin de l'exécution. Ces spécifications sont écrites souvent dans un langage basé sur la logique du premier ordre et dont la sémantique est totalement liée au langage de programmation avec lequel est écrit le code qu'on veut prouver. Si on prend le code suivant [15] qui fait la division de deux entiers positifs x et y (division de x par y).

{ y ≠0}
r:=x; q:=0;
while y<=r do
r:= r-y;
q:=1+q;
{(not (y<=r)/\ (x= r+y*q))}

Ce programme est spécifié en posant la précondition *(y ≠ 0)* et la post condition (*not(y<=r)/\ (x= r+y*q))* qui signifie qu'à la fin de l'exécution du code on obtient deux nombres *q* et *r* tels que *(x= r+y*q)*.

Les programmes et les annotations sont traités par des outils qui génèrent des conditions de vérification qu'il faut prouver pour se convaincre de la correction du programme. Dans ce cadre, plusieurs outils ont été développés, citons le système d'annotation JML (JAVA Modeling Language) [58] pour les programmes JAVA et l'outil Caduceus [06] pour les programmes C. Notons aussi que cette approche a été initialement consacrée pour la preuves des programmes impératifs (comme C ou Pascal), et très récemment [21], elle a été utilisée pour prouver des programmes fonctionnels.

4.2.2 La génération de spécification

Cette technique consiste à générer une spécification (ou un modèle) à partir d'un code source en effectuant des transformations préservant sa sémantique, puis de vérifier cette spécification par rapport à celle de l'utilisateur [53].

Le système $SOSSub_c$ [53] utilise cette technique : le programme source est transformé en un ensemble d'équations. Les spécifications de l'utilisateur sont également exprimées sous forme d'équations (en logique équationnelle). Ensuite, la preuve de la conformité du programme par rapport aux spécifications est établie : on doit arriver à prouver les équations de la spécification à partir des équations qui représentent le programme.

La génération de la spécification est aussi utilisée par la méthode Why [08] qui construit un modèle fonctionnel à partir d'un programme impératif annoté de sa spécification. Nous reviendrons dans le chapitre 4 sur le fonctionnement de cette méthode.

5. Les systèmes des preuves

Pour avoir donc des programmes certifiés, le système et ses propriétés peuvent être modélisés dans un système logique. Un système logique est associé à une théorie composée d'axiomes, qui sont des formules supposées vraies sans démonstrations et des règles d'inférences permettant de déduire une formule (la conclusion) à partir d'un ensemble de formules vraies (les prémisses). La preuve d'une formule consiste alors en une succession d'applications de règles du système, à partir des axiomes, permettant d'aboutir à la formule visée. J. RUSHBY [52] propose un classement à trois niveaux des démonstrations :

- Le premier niveau regroupe les outils fournissant un cadre formel à la démonstration sans pour autant en fournir un support. Les démonstrations y sont faites à la main et une preuve est validée lorsqu'elle convainc les évaluateurs.
- Les outils du deuxième niveau proposent en plus un système formel permettant une formulation rigoureuse de la démonstration. L'usage de la langue naturelle, possible au niveau précédent, disparaît. Les preuves sont cependant toujours menées manuellement.
- Le troisième niveau regroupe les outils intégrant la formulation d'une démonstration en leur cadre : les systèmes de preuve.

Des trois niveaux, le troisième propose, à travers les systèmes de preuves le plus haut niveau de rigueur et de garantie. Nous présentons donc les systèmes de preuves avec un intérêt particulier pour les assistants de preuves.

5.1 Définition

Les systèmes de preuve sont des logiciels issus de l'interaction des formalisations des mathématiques et de l'informatique. Ils permettent de profiter de la rapidité et de la rigueur des machines pour fournir aux utilisateurs un cadre pour écrire des théories mathématiques (axiomes, fonctions, théorèmes) ainsi que des spécifications de programmes et d'effectuer des preuves. Ces logiciels proposent des outils pour aider l'utilisateur (langage de spécification, bases de connaissances…) et surtout, ils lui garantissent la correction de son développement. Parmi les systèmes de preuves les plus connus citons : Coq [01], PVS [66], Isabelle [62], HOL [61], haRVey [64], et Simplify [63].

5.2 Critères de classification

Les systèmes de preuve peuvent être classifiés selon les critères suivants :

5.2.1 Critère de de Bruijn

Ce critère ([59], [47]) énonce que la correction des preuves dans le système doit être garantie par un programme de taille réduite (le vérifieur). Cela veut dire qu'il y a un noyau de preuve par lequel sont « filtrées » toutes les théories définies dans le système. Chacun des systèmes Coq, Isabelle et HOL a un noyau consacré à la vérification des preuves. En revanche, PVS, par exemple, n'a pas cette notion de noyau : de nouvelles méthodes de preuves peuvent être ajoutées et la correction des preuves nouvellement obtenues ne dépendra que de la correction de l'implémentation de la nouvelle méthode.

5.2.2 Logique traitée

Nous trouvons dans pour ce critère deux classes ([47], [59]) :

- les systèmes qui ont une logique fixe comme Coq (logique constructive avec théorie des types d'ordre supérieur), PVS (logique classique du premier ordre) ;
- les systèmes qui ont une méta – logique : c'est-à-dire une logique permettant de définir d'autres logiques. Le système Isabelle est basé sur une méta – logique et parmi les logiques développées pour ce système, on trouve une logique d'ordre supérieure (Isabelle /HOL) et une logique du premier ordre.

5.2.3 Automatisation/ Expressivité

Ces deux facteurs antagonistes ([60], [59] ont un impact important sur le comportement déductif des moteurs de preuves : d'une part l'expressivité de la logique permet l'énoncé de propriétés complexes, d'autre part, la simplicité du formalisme logique utilisé permet une grande automatisation. Le système Coq, par exemple, qui possède un langage de spécification très expressif a une faible automatisation.

5.2.4 Style d'interaction

Suivant le style d'interaction avec l'utilisateur, on distingue deux catégories de systèmes de preuve ([47], [59], [60]) :

Les démonstrateurs de théorèmes

Les démonstrateurs de théorèmes privilégient l'automatisation de la déduction au pouvoir d'expression. La preuve, une fois le système paramétré, est automatique. Le système haRVey est un démonstrateur de théorèmes qui traite de la logique équationnelle (les formules élémentaires sont des égalités) du premier ordre. A partir d'une paire constituée d'une formule est d'un ensemble d'axiomes, haRVey permet de construire une preuve de la formule donnée. Le système s'arrête si la formule soumise n'est pas prouvable à partir de l'ensemble d'axiomes.

Les assistants de preuves

Le choix des assistants de preuve est de privilégier l'expressivité du langage de spécification. Dans ce cas, les logiques en question sont d'ordre supérieur et donc indécidables. Il n'existe pas de procédure de décision construisant une preuve pour une propriété arbitraire. Si ce fait parait restrictif, il faut cependant souligner que beaucoup de propriétés ont besoin de la puissance de telles logiques pour être convenablement exprimés et démontrées.

Pour réaliser la preuve, l'intervention de l'utilisateur est nécessaire, et le procédé standard que l'on retrouve dans tous les assistant de preuve est la construction interactive de la preuve par l'application de commandes manipulant la preuve. Une telle commande est appelée « tactique ». Une tactique exprime un pas de preuve de base, cependant, elle peut permettre un raisonnement complexe ou même une preuve complète. C'est le cas avec les tactiques implémentant des procédures de décision pour un fragment décidable de la logique d'ordre supérieur.

Ces systèmes sont aussi appelés « vérifieurs de théorèmes » car ils fournissent à l'utilisateur les moyens de construire une démonstration, mais puisque l'utilisateur n'est pas à l'abri de l'erreur, ces systèmes doivent vérifier la validité de la preuve construite. Coq, PVS et Isabelle/HOL sont des assistants de preuves.

Les assistants de preuves sont utilisés dans plusieurs travaux de certification. Par exemple, le système Isabelle/HOL est utilisé dans la certification de programmes Java ainsi que dans le projet *Verisoft* qui ambitionne la vérification complète des systèmes informatiques incluant matériel, systèmes d'exploitations et applications utilisateurs [32]. L'assistant de preuve Coq est utilisé pour extraire des programmes Ocaml certifiés [03], prouver des programmes écrits en C, Java ou ML [07]. Il est également utilisé dans le cadre du projet *CompCert* dans le but de développer un compilateur certifié pour le langage C [65] et dans le projet *FOST* [80] dans le but de certifier des programmes de calcul scientifique.

CHAPITRE 2

Preuves des programmes impératifs
Et
Des programmes numériques

Chapitre 2
Preuves des programmes impératifs Et des Programmes numériques

1. Preuves des programmes impératifs

Le paradigme de programmation impérative correspond à une approche où les programmes sont vus comme des actions modifiant le contenu de la mémoire de la machine. L'action de base est l'affectation de variable. Chaque variable représente le contenu d'une case mémoire. L'affectation correspond à une opération qui modifie ce contenu. Ce paradigme est le plus proche du langage machine. Parmi les langages de programmation impératifs, citons: C, Fortran et Ada.

Pour effectuer des preuves sur les programmes impératifs, plusieurs techniques ont été développées, parmi lesquelles nous avons :

1) La logique de Hoare et la logique de Dijkstra ;
2) Les techniques de plongement;
3) Les techniques de raisonnement sur la mémoire.

1.1 La logique de Hoare

La logique de Hoare [33] est une discipline qui consiste à annoter les programmes avec des formules logiques, appelées « assertions », et à extraire d'autres formules logiques, appelées « obligations de preuve » à partir d'un tel programme annoté. La validité des obligations de preuve entraîne la correction du programme vis-à-vis des annotations : sa spécification formelle. Pour un programme (ou une partie d'un programme) *S*, on aura le triplet de Hoare suivant : *{P} S {Q}* où :

- *S* est l'action spécifiée;
- *P* et *Q* sont des formules logiques portant sur les variables du programme et appelées respectivement *précondition* et *postcondition* telles que: la précondition indique les conditions logiques qui doivent être satisfaites avant l'exécution de l'action S et la postcondition indique les conditions logiques vérifiées après l'exécution de S.

1.1.1 Correction d'un triplet de Hoare

Un triplet *{P} S {Q}* est ***partiellement correct*** si pour tout état initial vérifiant P, si l'exécution de S se termine, alors Q est vraie après l'exécution de S. Un triplet *{P} S {Q}* est ***totalement correct*** si pour tout état initial vérifiant *P*, alors l'évaluation de *S* se termine et l'état final des variables vérifie *Q* [32].

Le principe est d'utiliser un raisonnement déductif : le triplet *{P} S {Q}* est correcte s'il est dérivé comme conséquence de l'application des règles d'inférence et des axiomes de la logique de Hoare.

1.1.2 Les règles d'inférence

Les règles d'inférence présentées dans ce paragraphe correspondent aux constructeurs de base d'un petit langage de programmation à boucle *while* ([34], [33]) :

1.1.2 .1 L'instruction Skip

L'instruction *skip est l'instruction qui « ne fait rien »*. Elle joue le rôle d'instruction neutre. Elle est donnée par l'axiome *skip: {P} skip {P}.*

1.1.2 .2 L'affectation

L'axiome correspondant à l'affectation s'appelle *axiome de substitution arrière*. Il s'écrit de la manière suivante :

$$\{P\ [x \leftarrow exp]\}\ x := exp\ \{P\} \quad \text{(affect.)}$$

La notation *P [x←exp]* correspond à l'assertion P où toutes les occurrences de *x* ont été remplacées par *exp*. Cet axiome se lit de la manière suivante : dans l'état d'arrivé, *x* est égal à *exp*, donc pour qu'une propriété pour *x* soit vraie, il suffit qu'elle soit vraie pour *exp* dans l'état d'origine.

1.1.2.3 La composition séquentielle

La règle d'inférence correspondant à la composition séquentielle (enchaînement) de deux instructions est :

$$\frac{\{P1\}\ S\ \{P2\} \quad \{P2\}\ T\ \{P3\}}{\{P1\}\ S;T\ \{P3\}}\ \text{(seq.)}$$

Pour prouver *{P1} S ; T {P3}*, on doit trouver un prédicat *P2* tel que les deux triplets : *{P1} S {P2}* et *{P2} T {P3}* soient corrects.

1.1.2.4 La conditionnelle

La règle d'inférence pour la conditionnelle exprime le fait qu'il y a deux exécutions possible, selon que la condition est vraie ou fausse : si dans ces deux cas, lorsque *P* est vraie au départ et que l'on termine et on obtient *Q*, alors il en est de même pour l'instruction conditionnelle.

$$\frac{\{P \wedge b\}\ S\ \{Q\} \quad \{P \wedge \neg b\}\ T\ \{Q\}}{\{P\}\ \text{if b S else T}\ \{Q\}}\ \text{(cond.)}$$

1.1.2.5 La boucle

La dernière construction est une instruction d'itération de la forme *while (condition) do (instruction(s)).* Les preuves sur de telles constructions sont plus compliquées que les précédents du fait qu'il y a une infinité d'exécutions possibles [34].

La technique utilisées s'apparentent à de la récurrence : on montre qu'une propriété appelée *invariant (qu'on note Inv)* est vraie quand la boucle commence, est préservé à chaque itération et qu'à l'arrêt, l'invariant implique la postcondition. Cela s'exprime par la règle suivante :

$$\frac{P \Rightarrow Inv \qquad \{Inv \wedge b\}\ S\ \{Inv\} \qquad (Inv \wedge \neg b) \Rightarrow Q}{\{P\}\ \text{while b do (Inv)S}\ \{Q\}} \text{ (while)}$$

Remarque 1 (Les invariants des boucles)

Par le caractère même de la boucle, qui ne renvoie pas de résultat mais modifie des données en place, il se dégage le plus souvent une propriété qui persiste à chaque itération de la boucle : *l'*invariant *de boucle*. En effet, c'est l'idée de cet invariant qui guide le programmeur dans l'écriture de la boucle même si parfois cette idée reste informelle. Le choix d'un invariant est parfois difficile et il n'existe pas de méthode pour le trouver. On peut cependant appliquer les règles empiriques suivantes :

- Il y a dans le corps de la boucle des variables qui évoluent, un invariant est une relation entre ces variables, et éventuellement d'autres. Pour les variables de la boucle, il doit indiquer les valeurs des variables à la sortie d'un tour de boucle en fonction de leurs valeurs à l'entrée du tour.
- La définition de l'invariant est guidée par la postcondition : on doit écrit un invariant qui doit impliquer la postcondition.

Pour illustrer le cas de la boucle, considérons le triplet de Hoare *(T)* concernant un fragment de programme qui calcule la factorielle d'un entier *n* strictement positif [34] :

$$\{n > 0 \wedge i = n \wedge fact = 1\}\ \text{while}\ (i \neq 1)\ \text{do}\ fact := fact * i,\ i := i - 1, \{fact = n!\}\ \text{(T)}$$

Pour cet exemple, les variables modifiées par la boucle sont *fact* et *i.* En s'aidant des conditions initiales, ainsi que de la relation entre les variables à chaque itération, on voit qu'on a la relation *fact * i ! = n !* , pour les valeurs de *i* comprises entre *n* et *1*, et que pour la valeur 1, cette assertion correspond à la postcondition. C'est cette assertion qui constitue l'invariant pour la boucle. Suivant la règle *(while),* on doit démontrer:

$$\{i \neq 1 \wedge fact * i! = n!\}\ fact := fact * i,\ i := i - 1\ \{fact * i! = n!\} \quad \text{(T1)}$$

$$n > 0 \wedge n = i \wedge fact = 1 \Rightarrow fact * i! = n! \quad \text{(P1)}$$

$$\neg i \neq 1 \wedge fact * i! = n! \Rightarrow fact = n! \quad \text{(P2)}$$

Pour prouver (T1) qui correspond à une séquence, on utilise d'abord la règle (aff.) qui sert à établir l'assertion intermédiaire par une substitution en arrière dans l'assertion finale. Cela donne les deux sous – triplets suivants :

$$\{i \neq 1 \wedge fact * i! = n!\}\ fact := fact * I\ \{fact * (i - 1)! = n!\} \quad \text{(T1.1)}$$

$$\{fact * (i - 1)! = n!\}\ i := i - 1\ \{fact * i! = n!\} \quad \text{(T1.2)}$$

Pour les prouver, on applique la règle de l'affectation et du renforcement de la précondition. On se ramène à la preuve de:

$$i \neq 1 \wedge fact * i ! = n ! \Rightarrow fact * i * (i - 1) ! = n ! \quad (P3)$$

$$fact * (i - 1) ! = n ! \Rightarrow \{ fact * (i - 1) ! = n! \} \quad (P4)$$

On obtient les conditions de vérifications P1, P2, P3 et P4 dont les démonstrations vont faire appels à de l'arithmétique élémentaire et aux propriétés de la factorielle qui doivent avoir été énoncées dans la spécification.

Remarque 2 (Terminaison des boucles)

Il est possible de généraliser la logique de Hoare de manière à vérifier la correction totale d'un programme, c'est-à-dire sa correction partielle et sa terminaison. Pour cela il faut identifier un *variant de boucle* : un nombre naturel supérieur à zéro, écrit en fonction des variables de la boucle. Ce nombre décroît à chaque itération de la boucle jusqu'à atteindre une borne (le zéro généralement). Mais comme pour l'établissement d'un invariant, il n'y a pas de méthode générale pour déterminer le variant.

Pour une boucle annotée de son invariant *(Inv)* et de son variant *(v),* on aura la règle d'inférence suivante :

$$\frac{P \Rightarrow Inv \quad Inv \wedge b \Rightarrow v > 0 \quad \{Inv \wedge b \wedge v = v0\}\ S\ \{Inv \wedge v < v0\} \quad (Inv \wedge \neg b) \Rightarrow Q}{\{P\}\ \text{while b do (Inv,v) S}\ \{Q\}}$$

Le variant de boucle doit être supérieur à zéro. Si on commence un tour de boucle avec *(v =v0),* alors à la fin du tour, *v* devient inférieur à *v0.* Pour l'exemple précédent, le variant de la boucle est l'entier *i*.

1.1.2.6 Les règles logiques

Les axiomes et les règles permettant de raisonner sur les triplets de Hoare dépendent du langage de programmation considéré. Cependant, on a toujours les deux règles d'inférences suivantes qui correspondent respectivement au renforcement de la précondition et à l'affaiblissement de la postcondition :

$$\frac{P1 \Rightarrow P \quad \{P\}\ S\ \{Q\}}{\{P1\}\ S\ \{Q\}}\ (\text{renf.}) \qquad \frac{\{P\}\ S\ \{Q\} \quad Q \Rightarrow Q1}{\{P\}\ S\ \{Q1\}}\ (\text{aff.})$$

Le renforcement de la précondition indique que si en partant d'un état vérifiant *P, S* mène dans un état vérifiant *Q,* ceci est vrai pour tout état impliquant *P*. Et l'affaiblissement de la postcondition indique que si on a une preuve de *{P} S {Q}* et si on a une preuve que tout état vérifiant *Q* vérifie aussi une propriété *Q1,* on peut affirmer *{P} S {Q1}.*

1.1.3 Quelques travaux sur la logique de Hoare

Les axiomes et règles d'inférence données ci-dessous correspondent à un langage très simple qui a deux propriétés pour la validité de l'axiome de l'affectation :

- Les expressions *(exp)* dans l'affectation n'ont pas d'effets de bords, c'est-à-dire que leur évaluation ne modifie pas les variables du programme ;
- Il n'y a que des variables simples, c'est-à-dire des identificateurs, on n'a pas des éléments de tableaux par exemple.

Depuis le papier fondateur de cette discipline [33], où cette logique est présentée pour un langage impératif simple, plusieurs travaux ont été menés pour étendre ce formalisme et résoudre des difficultés, comme par exemple [47]:

- Pouvoir référer dans la postcondition Q d'un triplet {P} S {Q} à la valeur d'une variable avant l'exécution du programme S ;
- Avoir des effets de bords dans les expressions;
- supporter les break, continue et les exceptions;
- Avoir des structures de données complexes : tableaux, structures, pointeurs, objets... ;

Plusieurs travaux ont également formalisé la logique de Hoare pour extraire automatiquement des obligations de preuves à partir d'un programme annoté. Pour illustrer cette démarche, citons le travail de Norbert SCHIRMER [32] qui a formalisé la logique de Hoare dans le système de preuve Isabelle/HOL :

- L'auteur introduit d'abord un langage de programmation appelé Simpl (Sequential imperative programming language) qui est indépendant des langages de programmation concrets, mais assez expressif pour couvrir plusieurs caractéristiques de ceux-ci (les exceptions, les procédures mutuellement récursives, les instructions break...);

- Ensuite, une logique de Hoare pour les corrections partielle et totale est définie en donnant des règles d'inférence pour tous les constructeurs du langage de programmation. L'application des règles de la logique de Hoare est automatisée dans un générateur de conditions de vérification, implémenté comme une tactique (commande) appelé *vcg (verification condition generator)* dans Isabelle/HOL.

- Pour établir le lien avec un langage de programmation réel, une partie du langage C appelée C0 est formalisée dans le langage Simpl, de telle sorte que les propriétés prouvées pour un programme écrit en Simpl soient vérifiées également pour le même programme écrit en C0.

1.2 La logique de Dijkstra

La logique de Dijkstra a été développée après les travaux de Hoare. Dans cette approche, Dijkstra propose un calcul sur les assertions qui est ascendant. Il propose en partant d'une postcondition, de trouver *la plus faible précondition* (Weakest Precondition) qui vérifie le triplet de Hoare. Ainsi, on raisonne toujours sur des triplets, mais les règles sont établies dans l'autre sens dans un calcul ascendant (abduction) [24].

1.2.1 Définitions [29]

Soient A et B deux formules logiques. On dit que A et plus fort que B et inversement B est plus faible que A, Si A =>B.

Soit l'ensemble des formules suivant : $\{A_1, A_2, A_3, \ldots\}$. La formule A_i est la plus faible formule de l'ensemble si A_j => A_i pour toute formule A_j de l'ensemble. Par exemple, la formule $(y < 4)$ est la plus faible formule dans l'ensemble $\{(y \leq 3), ((y = 1) \vee (y = 3)), (y < 4))\}$.

Pour un programme S et une formule Q, WP (S, Q), la plus faible précondition pour S et Q est la plus faible formule P tel que : $\models \{P\}\ S\ \{Q\}$.

Proposition Un programme S est correct par rapport à une précondition Q et une postcondition Q si P implique la plus faible précondition pour S et Q :

$$\models \{P\}\ S\ \{Q\} \leftrightarrow \models P \Rightarrow WP\ (S, Q).$$

1.2.2 Propriétés du calcul WP

Le calcul de la plus faible précondition satisfait les propriétés suivantes [29] :

La propriété dite: « The excluded miracle »: WP (S, false) ≡ false.

La conjonction: WP (S, P) ∧ WP (S, Q) ≡ WP (S, P∧Q).

La disjonction: WP (S, P) ∨ WP (S, Q) ≡ WP (S, P ∨ Q).

La monotonie: $$\frac{P \Rightarrow Q}{WP(S,P) \Rightarrow WP(S,Q)}$$

1.2.3 Règles du Calcul WP

Voici les règles de calcul de la plus faible précondition attribuées aux constructeurs d'un mini langage de programmation à la Pascal ([29], [24]) :

1.2.3.1 L'instruction Skip

$$WP(Skip, Q) \equiv Q$$

La formule Q est valide après l'exécution de Skip si elle l'était avant l'exécution.

1.2.3.2 L'affectation

$$WP(x := t, Q(x)) \equiv Q(x)\left[x \leftarrow t\right]$$

Si la substitution des occurrences de x par t fait que Q(x) soit valides « maintenant », alors, après l'exécution de l'affectation (x := t), Q(x) sera valide car x aura la valeur t.

1.2.3.3 La composition séquentielle

$$WP(S; T, Q) \equiv WP(S, WP(T, Q))$$

Si la séquence (S ; T) doit établir la formule Q, alors T doit établir Q, et par définition, WP (T, Q) doit être vérifiée. C'est exactement la formule que doit établir l'exécution de S.

1.2.3.4 La conditionnelle

$$WP(\text{if } B \text{ then } S \text{ else } T, Q) \equiv (B \Rightarrow WP(S, Q)) \wedge (\neg B \Rightarrow WP(T, Q))$$

Si la condition B est vraie alors la plus faible précondition de la conditionnelle est WP(S, Q). Si la condition B est fausse alors la plus faible précondition de la conditionnelle est WP (T, Q).

1.2.3.5 La boucle

Pour calculer sa WP, la boucle doit être annotée de son invariant I est de son variant v.

$$WP(\text{while } B \text{ do } (S, I, v), Q) \Leftarrow I \wedge (I \wedge B \Rightarrow WP(S, I)) \wedge (I \wedge \neg B \Rightarrow Q) \wedge (I \Rightarrow v \in N) \wedge (I \wedge B \wedge v = v0 \Rightarrow WP(S, v < v0))$$

Pour le cas de l'itération, on calcule une précondition mais elle n'est pas forcément la plus faible car le calcul dépend de l'invariant que pose l'utilisateur. Ceci justifie le fait que pour cette règle nous avons une implication et non une équivalence.

1.2.4 Utilisation du calcul WP pour les preuves des programmes

Le calcul de la plus faible précondition est utilisé par plusieurs outils comme stratégie de vérification des programmes. Dans le système Theorema [81] par exemple, ce calcule est réalisé par une fonction appelée *EPT (Extended Predicate Transformer)* [35].

Afin de vérifier le programme S_1 ; S_2 ; … ; S_{n-1} ; S_n pour une précondition P et une postcondition Q, la fonction EPT est d'abord appelé pour calculer P_{n-1}= WP (S_n, Q). P_{n-1} devient une postcondition pour S_{n-1} et la fonction calcule alors WP (Sn_{-1} , P_{n-1}).

Le calcul se poursuit jusqu'à l'obtention de P_0. Il faut alors démontrer que P=>P_0.

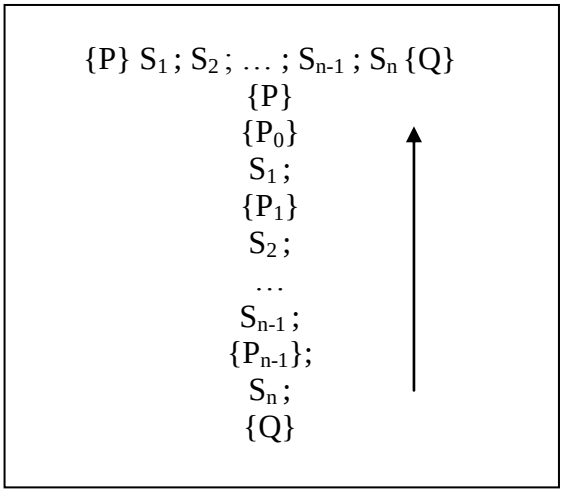

Figure 2.1 : Calcul de la plus faible précondition

En plus de P_0, la fonction EPT retourne également un ensemble de lemmes représentant les conditions de vérifications (Les invariants de boucle par exemple). Il faut également démontrer ces lemmes pour s'assurer de la correction du programme vis-à-vis de sa spécification.

Le calcul de la plus faible précondition est également utilisé par la méthode Why [07]. Why utilise le calcul WP pour extraire une spécification de chaque instruction du programme. Ces informations sont ensuite utilisées dans le cadre de la théorie des types pour générer des conditions de vérification.

1.3 Les techniques de plongement

Les techniques de plongement (embedding techniques) ont été introduites en 1992 dans le but d'utiliser des plateformes logiques existantes à des fins d'analyse formelle. Le principe général est de traduire un langage ou une logique source dans un langage ou une logique cible [22]. Leur utilisation dans le domaine de la preuve des programmes impératifs répond au besoin suivant : partant du fait qu'on cherche à prouver des propriétés issues d'un code annoté, il faut avoir un modèle formel du programme, et donc une formalisation du langage de programmation utilisé [09]. Il y a deux approches pour la formalisation des langages de programmation: le plongement superficiel (shallow embedding) et le plongement profond (deep embedding).

1.3.1 Formalisation par plongement superficiel

Cette technique consiste à fixer des notations conventionnelles pour traduire les constructions du langage en des constructions sémantiquement équivalentes dans le langage cible, seule la sémantique du langage est donc représentée. Au niveau raisonnement, le plongement superficiel est utilisé généralement par des outils de vérification de programmes comme l'outil Why. Ces outils prennent en entrée un programme spécifié dans le style de la logique de Hoare. Ils modélisent d'abord le programme par une sémantique donnée du langage. Ensuite, Ils construisent une preuve formelle que le modèle du programme vérifie la spécification ([09], [22]). Pour un exemple complet de plongement superficiel d'un langage de programmation impératif dans Coq, on pourra se référer à [49].

1.3.2 Formalisation par plongement profond

Cette approche consiste à traduire la syntaxe et la sémantique du langage source dans le langage/logique cible. Elle est utilisée pour des études méta- théoriques, c'est-à-dire sur le langage de programmation lui-même. Le plongement profond permet par exemple de prouver la propriété de la sûreté de typage (absence d'erreurs de typage à l'exécution) d'un langage de programmation. Parmi les travaux ayant suivi cette approche, on peut citer D. Von OHEIMB [48] où un sous ensemble du langage Java (Java light) a été formalisé dans le système de preuves Isabelle/HOL et la sûreté de typage a été prouvée pour ce langage. On peut aussi citer la formalisation du langage C dans le système HOL dans le travail de M.NORRISH [27].

1.4 Les techniques de raisonnement sur la mémoire

Le raisonnement sur les programmes impératifs nécessite la définition d'un modèle pour la mémoire, c'est à dire la description formelle de celle-ci et des opérations la manipulant. On distingue les techniques suivantes :

1.4.1 Le modèle concret

Ce modèle, le plus proche de la réalité, consiste à modéliser la mémoire comme un seul tableau d'octets. Son utilisation dans le cadre de la vérification des programmes s'avère très lourde: à chaque modification d'un emplacement de la mémoire, on doit prouver que les autres emplacements n'en ont pas été affectés [09]. De plus, certaines propriétés comme la séparation de zones mémoire ne sont pas exprimables. Ce modèle a été utilisé par N.MAGAUD [28] pour la vérification du programme de la racine carrée de la bibliothèque

GMP (GNU Multiple Precision Arithmetic Library) caractérisée par le fait que chaque argument pour les opérations GMP est décrit par la position de son mot mémoire de poids faible et le nombre de mots mémoire qui le composent. Ces informations aident à vérifier la validité des accès mémoire et à spécifier les zones mémoires modifiées.

1.4.2 Le modèle de Burstall-Bornat

Cette approche consiste à modéliser la mémoire comme une collection d'emplacements disjoints correspondant aux champs de structures et aux blocs alloués par des pointeurs dans un programme. Cette séparation assure qu'une modification dans un emplacement n'affecte pas les autres emplacements [09].

Ce modèle a été utilisé dans plusieurs travaux ([25], [26], [32]) pour prouver des propriétés de programmes manipulant les pointeurs et résoudre particulièrement des problèmes d'alias (Un *alias* arrive quand deux expressions distinctes désignent le même emplacement mémoire).

Ce modèle est utilisé par l'outil Caduceus [06]. Nous aborderons ce point au chapitre 4.

1.4.3 Logique de fragmentation

En plus des deux modèles précités, on peut citer aussi les travaux de J.C REYNOLDS qui a introduit un cadre de raisonnement sur des propriétés de séparation des zones mémoire appelée logique de fragmentation (separation logic) [31]. C'est une extension de la logique de Hoare qui permet de traiter les problèmes d'alias et de prouver des formules mettant en jeu des adresses et des blocs mémoire.

Un ensemble de commandes est d'abord introduit pour manipuler les structures de données : allocation, affectation, accès, et libération. Ensuite, un ensemble de connecteurs est défini. Par exemple, pour deux prédicats P1 et P2 :

- L'écriture P1 * P2 signifie que l'espace adressable peut être divisé en deux parties disjointes. Le prédicat P1 est vérifié dans une partie et le prédicat P2 est vérifié dans l'autre partie.
- L'écriture e->e_1 signifie que l'espace adressable contient une cellule à l'adresse e dont le contenu est e_1.

Des règles d'inférences sont définies pour les commandes manipulant les structures de données. Pour l'affectation, nous avons par exemple les deux règles suivantes :

$$\frac{}{\{e \hookrightarrow \_\ \}\ [e] := e_1\ \{e \hookrightarrow e_1\}} \qquad \frac{}{\{(e \hookrightarrow \_) * r\ \}\ [e] := e_1\ \{(e \hookrightarrow e_1) * r\}}$$

La logique de fragmentation est une discipline à part entière et plusieurs travaux lui ont donné suite. Cependant, son utilisation demeure difficile dans les systèmes existant du fait qu'elle introduit de nouveaux connecteurs [31].

2. Preuves des programmes de calcul numérique

Les structures de contrôle et les accès mémoires ne constituent qu'une partie de l'exécution d'un programme. Une certification complète doit prendre en considération les calculs numériques des programmes. Il est en effets indispensable de s'assurer que le résultat calculé par une application réponde bien au problème posé par l'utilisateur. L'obligation d'une preuve formelle est donc étendue aux calculs numériques [36].

Les travaux traitants des applications des méthodes formelles à ce type de programmes sont assez récents. Diverses raisons sont à l'origine de cet état de fait [12] :

1- Une première raison est l'utilisation importante des nombres réels dans les programmes numériques, alors que les méthodes formelles manipulent plutôt des nombres entiers et plus généralement des structures discrètes.

2- Une deuxième raison provient du fait que les méthodes formelles s'appliquent généralement aux programmes fonctionnels et elles sont, de ce fait, moins adaptées aux programmes d'analyse numériques, qui sont, le plus souvent, impératifs.

Le domaine des programmes numériques dans les méthodes formelles reçoit de plus en plus d'intérêt légitime. Plusieurs travaux ont été effectués dans cette optique. Nous pouvons les regrouper en trois catégories principales :

1) La formalisation basée sur l'arithmétique des ordinateurs ;
2) La formalisation de l'arithmétique d'intervalles ;
3) La formalisation des nombres réels.

2.1 Formalisation de l'arithmétique des ordinateurs

2.1.1 Arithmétique flottante IEEE-754

Les mémoires des ordinateurs sont limitées. Par conséquence, l'ensemble des nombres représentables en machine l'est également. On ne peut pas donc représenter tous les résultats de toutes les opérations possibles. L'arithmétique des ordinateurs permet alors d'étudier la manière d'effectuer de telles opérations [40].

Les nombres à virgule flottante sont, en général, les nombres utilisés dans les ordinateurs pour représenter les nombres réels. Les nombres flottants sont spécifiés par la norme IEEE-754 [41] qui a été définie pour satisfaire ou faciliter un certain nombre de propriétés dont en voici les principales :

- Conserver des propriétés mathématiques ;
- Faciliter la construction de preuves ;
- Rendre les programmes portables.

La plupart des processeurs actuels se basent sur la norme IEEE-754 pour leurs calculs flottants. Les nombres flottants IEEE-754 sont caractérisés par trois champs: une fraction ***f***, un exposant ***e*** et un signe ***s*** (qui prend la valeur 0 ou 1) :

Figure 2.2 : Nombre à virgule flottante IEEE-754.

La base de représentation étant fixée à 2. Ainsi, la valeur d'un nombre flottant ***x*** est calculée, sauf cas particulier, de la façon suivante:

$$(-1)^{s} \times 2^{e-B} \times 1.f$$

Où B est une valeur connue dépendant du format flottant, appelée le *biais*. Le nombre *(1.f)* est appelé *mantisse*.

En plus de la représentation des nombres flottants, la norme IEEE-754 permet de spécifier :

- Les différents formats pour les nombres. Par exemple: le format simple précision sur 32 bits et le format double précision sur 64 bits. Pour le format simple précision, l'exposant est représenté sur 8 bits, la fraction 23 sur bits et le biais vaut 127.
- les valeurs particulières comme les infinis et la valeur zéro. Pour le zéro signé par exemple, la valeur de la fraction est 0. Quant à l'exposant, il vaut e_{min}*-1,*tel que e_{min} est le plus petit exposant pour le format considéré.
- les modes d'arrondi qui permettent de trouver une représentation flottante pour le résultat d'une opération quand celui-ci n'est pas représentable en machine, ce qui est souvent le cas (exemple : arrondi vers le plus près, arrondi vers zéro) ;
- les exceptions comme les divisions par zéro et les débordements de capacité ;
- les conversions ;
- les opérations élémentaires sur les flottants et leurs propriétés (addition, soustraction, multiplication, division et racine carrée).

2.1.2 Formalisation de l'arithmétique flottante

La formalisation de l'arithmétique flottante a été réalisée dans plusieurs systèmes de preuves comme PVS [46], HOL/Light [45]. Pour le système Coq, Les travaux de M. DAUMAS, L. THIERRY et L. RIDEAU [39] ont donné lieu à une bibliothèque pour raisonner sur les nombres flottants. L'originalité de cette bibliothèque réside dans le fait qu'elle soit générique : elle ne traite pas une base ou un format particulier.

Un nombre flottant *(float)* est défini comme un enregistrement comportant deux champs *Fnum* et *Fexp*, comme suit :

Record float :Set: = Float {Fnum: Z; Fexp: Z}.

Le champ *Fnum* est un entier relatif *(Z)* représentant la mantisse du nombre flottant. Le champ *Fexp* est un entier relatif représentant l'exposant du nombre flottant. Le fait que *float* est de type *Set* indique que *float* est un type de donnée calculatoire.

Pour donner une sémantique à ce nouveau type, une fonction *FtoR* est définie pour calculer la valeur numérique d'un nombre flottant :

*Definition FtoR (x: float):= (Fnum x * (powerRZ (IZR radix) (Fexp x))) %R.*

La valeur numérique d'un flottant *x* est le produit signé dans R, les réels de Coq, *(%R)* de sa mantisse par la base *(radix)* élevée à la puissance *(powerRZ)*de son exposant. La base est un entier supposé supérieur à 1.

A partir de là, les auteurs définissent plusieurs notions : l'égalité sur les flottants, la définition des opérations arithmétiques sur le type *float* ainsi que des propriétés sur les opérations, les nombres représentables en machine et les modes d'arrondi.

Cette bibliothèque permet d'explorer un grand nombre de propriétés mathématiques associées à la l'arithmétique flottante et de garantir la validité de méthodes parfois subtiles employées en calcul flottant et dans l'arithmétique multiprécision [40] ainsi que pour la vérification du comportement flottant de programmes C [30].

Il serait intéressant de parler également du travail de S. BOLDO et J-C. FILLIATRE [30] Les auteurs y présentent une extension d'un outils de vérification de programmes C : Caduceus. L'extension présentée permet d'insérer dans les programmes C, selon le style de la logique de Hoare, des annotations portant sur les propriétés des nombres flottants et de les prouver. Ce travail, utilisé avec la librairie précitée a permis de démontrer plusieurs programmes.

Voici un exemple [30] permettant d'illustrer ce travail : La fonction C suivante permet de démontrer le théorème de Sterbenz qui stipule que si *x* et *y* sont deux flottants représentables exactement en machine (sans erreur d'arrondi) tels que $(y/2) \leq x \leq (2*y))$, alors le résultat de la soustraction sur les flottants *(x-y)* est exactement représentable.

```
/*@ requires y/2 <= x <= 2*y
@ && \round_error(x) ==0
@ && \round_error(y) ==0
@ ensures
@ \round_error(\result)==0
@*/
double Sterbenz(double x,double y){
return x-y;}
```

En plus des déclarations et du corps de la fonction, nous avons une précondition *(requires)* qui énonce que $(y/2) \leq x \leq (2*y))$ et qu'il n' y a pas d'erreur d'arrondi sur les valeurs *x* et *y (\round_error(x) =0 et \round_error(y) =0).* La post condition *(ensures)* signifie qu'il n' y a pas d'erreur d'arrondi sur le résultat *(\result)* retournée par la fonction.

A partir de ce programme, des obligations de preuves sont générés, elles sont prouvées en utilisant l'extension de l'outil Caduceus et la bibliothèque des nombres flottants.

2.2 Formalisation de l'arithmétique d'intervalles

L'arithmétique d'intervalles consiste à remplacer un nombre réel par un intervalle qui le contient. On calcule sur les intervalles plutôt que sur les valeurs et il est garanti que le résultat est contenu dans l'intervalle calculé. Elle a été introduite dans les calculs sur ordinateurs par Ramon E. Moore en 1962 [44]. De nombreuses applications ont été développées depuis son introduction comme l'optimisation linéaire et la résolution d'équations différentielles ordinaires.

L'arithmétique d'intervalles a été formalisée dans l'assistant de preuve PVS dans le but de pouvoir justifier formellement des calculs qui apparaissent dans la vérification d'applications scientifiques comme la résolution de conflit de navigation aérien [37].

Nous ne pouvons parler d'arithmétique d'intervalles sans citer l'outil GAPPA (Génération Automatique de Preuves de Propriétés Arithmétiques) [36]. C'est un logiciel de certification d'application effectuant des calculs numériques qui est basé sur l'arithmétique d'intervalle ainsi qu'un ensemble de théorèmes et de règles de réécriture.

Pour expliquer le principe de GAPPA, considérons l'exemple [36] de la fonction f, en C :

```
float f (float x) {
assert (0 <= x && x <= 1);
float y = x * (1 - x);
return sqrt(0.25 - y);}
```

Cette fonction prend en argument un flottant simple précision (*float*) x. L'assertion à son début spécifie que x est compris entre 0 et 1. La fonction calcule ensuite la valeur de y; il s'agit du flottant x * (1-x). Ce ne sont pas ici la multiplication et la soustraction sur les nombres réels ; il s'agit d'opérateurs spécifiques liés à l'arithmétique flottante, et comme les formats flottants sont limités, chaque valeur calculée est potentiellement entachée d'une erreur d'arrondi pouvant provoquer des comportements inattendus (division par zéro, racine d'un nombre négatif).

Pour garantir que ces différents problèmes ne risquent pas de se produire, il faut pouvoir prouver que chacune des valeurs calculées satisfait les contraintes de domaine qu'imposent les parties du programme qui utilisent ces valeurs.

Alors que l'on sait que la valeur $(1/4) - x * (1 - x) = (x - 1/2)^2$ est supérieur à zéro pour les nombres réels, il n' y a, à priori, aucune garantie que le nombre le soit pour les flottants.

Pour garantir qu'aucun comportement exceptionnel ne se produise, il faut prouver que la fonction ne cherchera pas à prendre la racine d'un nombre négatif. Si on note *F*, l'ensemble des nombres flottants simple précision, cela revient à prouver la proposition logique suivante :

$$\forall\ x \in F,\ 0 \leq x \leq 1 \rightarrow 0.25 - (x * (1 - x)) \geq 0$$

Dans cet exemple, la certification du bon comportement d'une fonction a été ramenée à la preuve d'une proposition contenant des encadrement d'expressions arithmétique. Ce genre de proposition est à la base des certifications que GAPPA effectue : à partir d'une proposition logique, il génère une preuve formelle de la propriété. Cette preuve peut être

vérifiée automatiquement par l'assistant de preuve Coq. Elle peut être aussi utilisée au sein d'une preuve plus générale qui prend en compte, en plus des erreurs de calculs, la validité des accès mémoire et la terminaison des boucles par exemple.

L'outil GAPPA a été utilisé pour la certification dans différents domaines, par exemple, les fonctions de la bibliothèque CRlibm telle que la fonction exponentielle [36]. Un rapprochement entre GAPPA et d'autres outils de preuve de programmes comme l'outil Why est également réalisé [78] pour une certification complète des programmes numériques [36]. Il convient de citer cependant, le principal inconvénient de l'arithmétique d'intervalles : l'intervalle retourné par le calcul peut être trop large, ce qui rend la solution impertinente [38].

2.3 Formalisation des nombres réels

2.3.1 Les erreurs numériques

Dans les langages de programmation qui servent à faire des calculs, on trouve donc les nombres à virgule flottante qui sont une approximation plus ou moins satisfaisante des nombres réels. On découvre donc en plus des résultas erronés, certaines propriétés élémentaires du corps des nombres réels non respectées [12].

La suite (a_n) suivante est un exemple ([12], [38]), des problèmes liées aux erreurs d'arrondi :

$$a_0 = 11/2, a_1 = 61/11,\ a_{n+1} = 111 - \frac{1130 - 3000 / a_{n-1}}{a_n}$$

Bien que la notion de calcul dans un langage de programmation n'est pas une preuve, on peut tout de même espérer trouver un résultat assez proche de celui donné par une preuve mathématique. Pour cette suite, il a été démontré que :

$$a_n = \frac{6^{n+1} + 5^{n+1}}{6^n + 5^n}$$, dont la limite à l'infini vaut 6.

En effectuant les calculs, en définissant une fonction récursive en Ocaml, on obtient les résultats suivants :

n	2	14	15	16	17	18	19	20	21	25	26
a_n	5.59	15.41	67.47	97.14	99.82	99.98	99.99	99.99	99.99	100.	100.

D'après ces résultats, on aurait tendance à croire que cette suite converge vers 100 et non vers 6. L'exemple suivant [12], en Ocaml, montre que la propriété de l'associativité de l'addition n'est pas respectée :

```
# (10000003.+. (-10000000.)) +. 7.501 ;;
  - : float = 10.501
# 10000003.+. (-10000000.) +. 7.501) ;;
  - : float = 10.5010000002
```

Ce qui signifie que *(10000003+ (-10000000)) + 7.501 ≠ 10000003+ (-10000000 + 7.501).*

Les problèmes liés aux résultats erronés ont mis en avant plusieurs solutions dont les systèmes de calcul formels qui offrent plusieurs formes pour définir les nombres réels. Cette approche consiste à avoir, au dessus de l'arithmétique des machine, une couche, logicielle, permettant de manipuler les nombres flottants en précision arbitraires : dans le système MAPLE [43], par exemple, l'utilisateur peut exiger que tous les calculs soient effectués avec une précision de 1000 chiffres décimaux.

Néanmoins, même si ces systèmes sont plus valides que les langages de programmation, ils ne permettent pas la vérification. En effet, l'exemple concernant la suite (a_n) reste valable, c'est-à-dire qu'elle continue à converger vers 100 dans les systèmes de calcul formel [12].

D'autres travaux ont plaidé pour le développement d'une arithmétique exacte [38] qui permet ou d'obtenir des résultats « aussi proche que l'on veut ». Pour cela, l'utilisateur doit indiquer une borne supérieure sur l'erreur d'arrondi autorisée sur le résultat final et cette borne est respectée tout au long du calcul.

Le principe de l'arithmétique exacte est de proposer des algorithmes permettant de représenter les nombres réels. On peut par exemple représenter un réel par des fractions continues, où par des entiers dans une base [38].

Le principe de l'arithmétique exacte est repris par certains travaux ayant pour but la formalisation des réels dans les systèmes de preuves afin d'établir des preuves de programmes travaillant sur les nombres réels.

2.3.2 Formalisation des nombres réels dans les systèmes de preuves

Dans ce paragraphe, nous présentons les principales approches possibles pour formaliser les nombres réels dans les assistants de preuves. La formalisation des réels est un travail qui soulève d'importants problèmes théoriques et généralement, on est contraint des faire un choix suivant les deux sources d'opposition suivantes [12] : Construction Vs axiomatisation, formalisation classique Vs formalisation intuitionniste.

2.3.2.1 Construction versus axiomatisation [12]

Construire les nombres réels

Les deux formalisations les plus connues sont la *méthode de Cantor* et *les coupures de Dedekind* dont voici une brève présentation :

Méthode de Cantor Cette méthode identifie un nombre réel à une suite de nombres rationnels convergente (vers ce nombre) en utilisant la propriété suivante. Un suite (S_n) converge vers s si :

$$\forall \varepsilon > 0, \exists \mathrm{N} \, \forall \mathrm{n} \geq \mathrm{N}, \left|\mathrm{S}_{\mathrm{n}} - s\right| < \varepsilon$$

Coupures de Dedekind Cette méthode identifie un nombre réel à un ensemble de nombres rationnels plus petits que lui. Si nous nommons S cet ensemble, il s'agit d'une coupure s'il vérifie les propriétés suivantes:

1. $\exists\ x, x \in S$
2. $\exists\ x, x \notin S$
3. $\forall\ x \in S, \forall\ y < x, y \in S$
4. $\forall\ x \in S, \exists\ y > x, y \in S$

Pour plus de détail concernant les deux méthodes, on pourra se référer à [42].

Axiomatiser les nombres réels

Axiomatiser une théorie revient à poser un ensemble minimal d'axiomes qui permettent de la spécifier. Une axiomatisation permet d'arriver rapidement aux aspects plus intéressants et recherchés d'un développement, à condition, bien entendu, que les axiomes posés ne suscitent pas le moindre doute. Ce dernier point peut être vu comme le point négatif de cette méthode, puisqu'un axiome faux permettrait de rendre toute proposition prouvable. Néanmoins, le fait que la théorie des nombres réels est bien connue rend le risque d'une formalisation, où un ensemble minimal de définitions ne serait pas prouvé faible.

La formalisation des réels dans la distribution Coq standard est une axiomatisation. Nous reviendrons sur ce point au chapitre 3.

2.3.2.2 Formalisme intuitionniste versus formalisme classique [12]

Une formalisation constructive ne peut être faite que dans un système de preuves basé sur la logique intuitionniste comme Coq. L'avantage d'une telle formalisation est qu'il est alors possible d'utiliser l'information calculatoire provenant des preuves effectuées. Cet aspect prend toute son importance lorsqu'on veut utiliser le principe de l'extraction. Cependant, dans certains cas, une formalisation intuitionniste peut s'avérer trop restrictive. Notamment, abandonner le tiers exclus, revient à abandonner des résultats importants en analyse.

Une formalisation classique est indispensable lorsqu'il est nécessaire de montrer l'existence de fonctions non calculables. De ce fait, s'il s'agit, par exemple, de montrer des théorèmes d'analyse mathématique, une telle formalisation semble mieux adaptée. L'inconvénient majeur est, bien entendu, de ne pas pouvoir profiter de l'avantage offert par la formalisation intuitionniste, à savoir l'extraction de programmes.

Le tableau suivant [12] permet de résumer les différents points concernant les choix pour la formalisation des réels. Les avantages sont notées par des « + » et les aspects négatifs par des «- ». Les considérations de temps (long, rapide) concernent la rapidité d'implémentation des propriétés de base, tandis que les considérations de difficulté (compliqué, simple) concernent aussi bien l'implantation que l'utilisation.

		Intuitionniste	Classique
Construction	+	Prouvé/extractible	prouvé
	-	Long/compliqué	long
Axiomatisation	+	Rapide +/-extractible	Rapide/simple
	-	Non prouvé/compliqué	Non prouvé Non extractible

Tableau 2.1 : Méthodes de formalisation des réels

2.3.3 Quelques exemples de formalisation des nombres réels

Dans ce paragraphe, nous allons citer quelques exemples de formalisation des nombres réels dans différents systèmes de preuves :

- Dans le système HOL, il s'agit d'une construction classique utilisant la méthode de Cantor faite par John Harrison en 1995 [42]. Une grande partie de l'analyse standard y est formalisée.
- Dans le système PVS, il s'agit d'une axiomatisation classique. Une partie de l'analyse réelle et des fonctions transcendantes y sont formalisées [79].
- Dans Mizar, il s'agit d'une construction classique basée sur les coupures de Dedekind [12].
- Dans Coq, la formalisation standard est une axiomatisation classique réalisé par M .MAYERO [12]. Il existe également une axiomatisation intuitionniste faite par M. NIQUI [19].

Les travaux de formalisation des nombres réels sont utilisés pour prouver des théorèmes issues de plusieurs domaines. La formalisation intuitionniste de M. NIQUI, par exemple, entre dans le cadre de l'extraction d'une preuve pour le théorème FTA (Fundamental Theorem of Algebra) [19]. La formalisation axiomatique dans Coq est utilisée pour prouver des programmes d'analyse numériques (projet CerPAN [77]) dans le cadre de la méthode Why par exemple.

CHAPITRE 3

L'assistant de preuves Coq

Chapitre 3

L'assistant de preuves Coq

1. Introduction

L'assistant de preuves Coq est basé sur le lambda-calcul et la logique constructive d'ordre supérieure.

1.1 Le lambda-calcul

Le lambda-calcul, introduit par A. Church [76], est une théorie des fonctions et constitue le formalisme de base pour plusieurs systèmes de preuves. Le lambda-calcul (λ – calcul) est caractérisé par la syntaxe suivante :

- Variables = {*a, b, c, d, …*}. Toutes les variables sont des λ -expressions, on peut les noter simplement *a* ou bien (*a*) ;
- Si *U* et *V* sont deux lambda-expressions alors *(U V)* est une λ -expression. Le terme *(U V)* peut être vu comme la valeur rendue par la fonction *U* Appliquée à l'argument *V* ;
- Si *a* est une variable, et *U* une λ -expression, alors *(λ a.U)* est une λ-expression, telle que a s'appelle un paramètre de la λ -expression. La λ -expression *U* est appelée le corps de *(λ a.U).*

La théorie du λ-calcul est démontrée équivalente aux machines de Turing. Toutes les fonctions calculables sont λ-définissables (Thèse de Church) [76].

Au λ-calcul, on peut associer un système de types. A chaque terme correspond un type qui détermine certaines propriétés. Le couple (λ-calcul, système de types) a été exploité par Curry et Howard en considérant les types comme des propositions et les λ-termes correspondants sont les preuves. Cette correspondance est connue sous le nom de *l'isomorphisme de Curry-Howard* [76].

1.2 La logique constructive

La logique constructive [03] est une logique où la loi du tiers exclu *(A \/ (non A))* n'est pas valable. Le but de cette logique n'est pas de savoir si une telle proposition est vraie ou non, mais son intérêt est de savoir si une telle proposition possède ou non une preuve. La preuve de l'existence d'un objet donne une méthode pour le construire. La valeur de vérité du calcul propositionnel dans la logique classique peut être obtenue à l'aide des tables de vérité. Dans la logique constructive, nous avons besoin d'introduire une sorte de calcul pour les preuves.

1.3 La sémantique de Heyting et Kolmogorov

La sémantique de Heyting utilise la notion de preuve comme objet. Les preuves sont considérées comme des objets manipulables au même titre que les entiers ou les fonctions. La sémantique de Heyting-Kolmogorov est décrite par [03] :

- Une démonstration A => B s'exprime comme une fonction qui associe à toute démonstration de A une démonstration B.
- Une démonstration de $\forall x \in A.P(x)$ est une fonction qui prend en argument un élément quelconque de A et rend une preuve de P (x).
- Une démonstration de $A \wedge B$ est un couple formé d'une démonstration de A et d'une démonstration de B.
- Une démonstration $P_{A \vee B}$ de $A \vee B$ a deux formes possibles :
  a) Soit $P_{A \vee B} = < g, P_A >$ où P_A est une preuve de A,
  b) Soit $P_{A \vee B} = < d, P_B >$ où P_B est une preuve de B.
  Les membres *g* et *d* permettent d'indiquer de quelle propriété le membre droit est une preuve. On voit ici que pour montrer $A \vee B$, il faut soit savoir prouver A, soit savoir prouver B.
- Une démonstration de $\exists x \in A.$ P(x) est une paire < a, h >, où a est un élément particulier (le témoin) de A et h une preuve de P (a). Cela veut dire que pour prouver une formule existentielle, il faut exhiber un témoin.

2. Le système Coq

Coq est un système d'aide à la preuve développé depuis 1986 à l'INRIA. Il est dû à l'origine à Thierry Coquand et Gérard Huet [01], et a profité de nombreuses contributions. Il est basé sur une logique d'ordre supérieure appelée calcul des constructions inductives. Il s'agit d'un lambda calcul typé basé sur une logique d'ordre supérieur avec types dépendants, types récursifs et polymorphisme de type. Les composantes essentielles du système Coq sont:

- Un noyau de vérification de types et de construction d'environnement bien typés;
- Un langage de spécification et de développement de théories mathématiques: Gallina;
- Un outillage d'aide à la construction interactive de preuves par des tactiques de preuve ;
- Un extracteur de programme: ce module de Coq permet d'extraire un programme sans erreur à partir de la preuve de sa spécification.

3. Le langage de Spécification : Gallina

Le langage de spécification du système Coq, Gallina ([01], [03]), nous permet de développer des théories mathématiques et de prouver des spécifications de programmes en définissant des objets tels que les axiomes, les théorèmes, les fonctions et les prédicats en offrant un ensemble de commandes qui permettent la déclaration et la spécification des différents objets.

Ce langage repose sur le CCI. Dans ce calcul tous les objets ont un type. Il y a des types pour les fonctions (ou les programmes), les données, les preuves et les types eux-mêmes possèdent un type. Par exemple, on ne permet pas la déclaration "pour tout x, P", on doit dire au lieu de cela : "Pour tout *x* appartenant à *T, P*". L'expression "*x* appartenant à *T*" est écrite

"x:T ". On dit aussi : "x de type *T*". Les termes de Gallina constituent une catégorie syntaxique très générale. Elle correspond à la notion intuitive d'expression bien formée prenant en compte les règles de construction du langage, par exemple, nous avons les expressions correspondant aux programmes fonctionnelles et les types.

Dans le système Coq, les types sont considérés comme des termes, or chaque terme dans Coq possède un type. Le type d'un type est appelé: *sorte*. Les sortes sont au nombre de trois:

- *La sorte Set*: elle contient les objets ayant un trait calculatoire, par exemple: l'ensemble des naturels.
- *La sorte Prop*: elle contient tout ce qui a trait à la logique par exemple les justifications et les pré/post-conditions.
- *La sorte Type*: C'est le type de Set et Prop. De plus, il y a une hiérarchie de types (i) pour n'importe quel nombre i. (cela empêche que « Type soit son propre type »). L'ensemble S des sorte est défini donc par: S = {Prop, Set, Type (i)/ i ∈ N}. Les sortes ont les propriétés suivantes: Prop:Type (0), Set: Type (0) et Type (i) : Type (i+1). Notons aussi que les indices sont invisibles à l'utilisateur.

3.1 Les types inductifs

Un type inductif ([02], [04]) est spécifié par la donnée de son nom, son types et les noms et types de ses constructeurs. Les constructeurs d'un type inductif *T* sont des fonctions dont le type final doit être *T* (ou une application de *T* à des arguments lorsque le type *T* est une fonction).

3.1.1 Les types inductifs sans récursivité

Les types inductifs les plus simples sont les types énumérés, utilisés pour décrire des ensembles finis. Nous donnons l'exemple de l'ensemble des valeurs booléennes qui contient deux éléments et qui est défini dans le système Coq comme suit:

Coq < Inductive bool:Set:=true:bool | false: bool.

Le type déclaré ainsi a pour nom *bool*, son type est *Set* et ses deux constructeurs sont *true* et *false.*

Les types inductifs peuvent aussi être utilisés pour représenter des agrégations de données pour définir des enregistrements. Dans ce cas, on utilise des définitions inductives à un seul constructeur. Ce constructeur a le type d'une fonction qui prend autant d'arguments qu'il y a de champs dans l'enregistrement. L'explication est que le constructeur est une fonction qui retourne un enregistrement si on lui donne les valeurs de tous les champs. Par exemple, un point dans un plan est décrit comme un couple de coordonnées. Si l'on s'intéresse à l'ensemble constitué de tous les points à coordonnées entières, le type correspondant pourra être décrit de la façon suivante:

Coq < Inductive plan:Set:=point:Z->Z->plan.

Où *Z* est l'ensemble des entiers dans Coq. Le constructeur point qui a le type *Z->Z->plan*, prend deux arguments dans *Z* qui représentent les deux champs du type enregistrement concerné.

3.1.2 Les types inductifs avec récursivité

Les types inductifs sans récursivité permettent de décrire des données dont la taille est connue d'avance. Cependant, il est nécessaire de pouvoir raisonner sur des structures de données dont la taille est non définie à l'avance. La récursivité fournit la solution et ceci en

exprimant que certaines données comportent des fragments de même nature que ces données elles-mêmes.

Pour bien illustrer ce fait, prenons l'exemple du type des nombres naturels qui est décrit dans Coq selon la définition suivante:

```
Inductive nat: Set :=  O : nat | S : nat -> nat.
nat is defined
nat_rect is defined
nat_ind is defined
nat_rec is defined
```

Cette définition introduit le type nat dans la sorte Set. Ce type possède deux constructeurs qui traduisent les deux manières dont sont construits les nombres naturels: soit on prend le nombre 0 (le constructeur O), soit on applique la fonction successeur à un nombre déjà construit (le constructeur S) et c'est dans le deuxième constructeur ou l'on trouve la récursivité. Par exemple le nombre naturel (S (S 0)) contient le fragment (S 0) qui est lui même un nombre naturel.

Nous remarquons que pour le type nat défini dans la paragraphe précédent, le système annonce trois définitions: *nat_rect, nat_ind* et *nat_rec.* En effet, pour tout type inductif T, le système Coq génère *automatiquement* trois définitions T*_rect, T_ind* et *T_rec* qui sont des théorème qui vont nous permettre de raisonner et de faire des calculs sur les données de ce type.

```
Coq < Check nat_ind.
nat_ind
     : forall P : nat -> Prop,
       P O -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
```

Le théorème *nat_ind* est appelé principe d'induction associé à la définition inductive. Il exprime que pour une propriété *P* donnée sur les données de type nat, si l'on arrive à démontrer qu'elle est satisfaite par *0* et que si l'on prend un nombre *n* qui la satisfait alors le nombre *(S n)* la satisfait aussi alors, tout nombre naturel satisfait *P*.

3.1.3 Vecteurs et matrices

Concernant les vecteurs, la distribution standard Coq fournit le type inductif défini comme ceci :

```
Inductive vector (A : Type) : nat -> Type :=
Vnil: vector A 0
| Vcons : A -> ∀ n: nat, vector A n -> vector A (S n).
```

Si *A* est un type et *n* un nombre naturel, alors *« vector A n »* est un vecteur dont le type des éléments est *A* et sa longueur est *n*.

Concernant le type matrice, il n'est pas fournit par la distribution standard. Plusieurs travaux existent à titre de contributions [01]. Citons :

- la contribution de J-C FILLIATRE qui définit le type matrice en se basant sur un algorithme de C. OKASAKI [17] ;
- La contribution N.MAGAUD qui définit une matrice comme un vecteur de vecteurs [18].

3.2 Les fonctions récursives

Une fonction se définit généralement en indiquant la valeur qu'elle retourne pour un paramètre donné. On dit: *la fonction qui à x associe l'expression e*. La variable x est autorisée à apparaître dans l'expression e, ce qui fait que la fonction n'est pas constante.

Pour une fonction récursive, on dit: *la fonction f qui à x associe l'expression e avec la possibilité que f apparaisse dans e.* En d'autres termes, on suppose que la fonction f déjà partiellement définie lorsqu'on essaye de déterminer la valeur qu'elle retourne pour une nouvelle valeur du paramètre.

Pour définir une fonction récursive dans Coq ([02], [04]), l'utilisateur doit déterminer les valeurs prises par la fonction dans un ordre précis qui suit l'ordre dans lequel sont construits les termes des types inductifs. Pour les nombres naturels, par exemple, on est obligé de construire *0* avant *(S 0), (S 0)* avant (S (S 0)) et ainsi de suite. Pour la définition d'une fonction récursive sur les naturels, on utilise la valeur (*f 0)* pour définir la valeur *(f (S 0)),* la valeur *(f (S 0))* pour définir la valeur *(f (S (S 0)))* et de manière générale, on utilise la valeur *(f n)* pour définir la valeur *(f (S n)).*

De point de vue pratique, la construction des fonctions récursives se fait dans Coq avec la commande *Fixpoint.* Par exemple, soit la fonction récursive *plus* qui calcule la somme de deux nombres naturels n et m :

```
Fixpoint plus (n m:nat) {struct n} : nat :=
  match n with
  | O => m
  | S p => S (plus p m)
  end.
```

Quand on définit une fonction récursive à plusieurs arguments, l'argument sur lequel la récursivité s'organise est appelé: *argument principal*. Dans la fonction *plus,* l'argument principal est *n*, ce qui est indiqué par l'écriture *{struct n}*. L'écriture « *match n with* » indique un filtrage par motif sur l'argument n.

3.2.1 Les fonctions structurellement récursives [04]

En pratique, les fonctions récursives définies avec *Fixpoint* contiennent toujours un filtrage sur l'argument de la récursivité. La description de la fonction est organisée suivant la *structure* du type inductif. Pour l'exemple précédent, comme il y a deux constructeurs dans le types inductif nat, on retrouve deux cas dans les deux fonctions; le deuxième constructeur a un argument dans le même type inductif, on peut donc utiliser des appels récursifs sur cet argument. On parle de *récursivité structurelle.*

Dans la récursivité structurelle, la terminaison de la fonction est vérifiée syntaxiquement auprès de l'argument principal des appels récursifs, en s'assurant qu'il décroît à chaque appel récursif. En effet, lors de l'appel récursif, la fonction se rappelle avec un argument qui est un sous terme de l'argument traité.

3.2.2 Définir des fonctions par récurrence bien fondée [04]

Cette méthode consiste à définir la fonction par rapport à une relation *R* bien fondée. Une relation bien fondée est une relation qui ne contient pas de chaîne infinie décroissante (l'ordre naturel sur les entiers positifs est un exemple de relation bien fondée).

L'idée est qu'une définition récursive est correcte si tous les appels récursifs se font sur des arguments « plus petits » que l'argument initial selon une certaine relation *R* bien fondée sur un type inductif *A*. La propriété de bonne fondation de la relation assure qu'il ne peut y avoir des suites infinies décroissantes d'appels récursifs de la fonction, ce qui assure sa terminaison. L'utilisateur doit fournir la preuve de la bonne fondation de la relation (si elle n'existe pas dans Coq) ainsi que des théorèmes qui assurent que l'argument de l'appel récursif est plus petit que l'argument initial.

Considérons, par exemple, la fonction *log* (définit sur le type nat) dont la description est la suivante:

log 0 = 0

log 1 = 0

log (S (S m))= S (log (div2 (S (S m)))) où *div2* est une fonction qui effectue la division sur 2 pour *nat.*

On remarque que lors de l'appel récursif, la fonction se rappelle avec l'argument *(div2 (S (S m)))* qui n'est pas un sous terme de l'argument traité *(S (S m)).* Cette fonction est donc non structurellement récursive (sa terminaison n'est pas vérifiée syntaxiquement). C'est dans ce genre de cas qu'il faut utiliser la récursivité bien fondée. La relation considérée est la relation *lt* : *inférieur à*, définie sur le type nat et qui est bien fondée. Il s'agit donc de démontrer que les arguments d'appels récursifs sont reliés par cette relation.

3.3 Les prédicats inductifs

Le mécanisme des définitions permet la définition d'objets logiques : les prédicats inductifs (ou propriétés inductives) [02]. On peut par exemple définir le prédicat *even* (être pair) sur l'ensemble des nombres naturels comme ceci :

```
Inductive even: nat ->Prop: =
        | even_0: (even 0)
        | even_ss: forall (n: nat), (even n) -> even (S (S n)).
```

La sorte *Prop* souligne l'intention logique de cette définition qui introduit le prédicat *even* avec les deux constructeurs *even_0* et *even_ss* qui définissent les clauses de *even* : zéro est pair et si n est pair alors n+2 (le successeur de son successeur) est aussi pair.

Les prédicats inductifs ont un très grand pouvoir expressif qui nous permet de formuler de nombreuses propriétés des données et des programmes (spécifier). En particulier, nous pouvons représenter des fonctions comme des relations inductives en vue de faciliter le raisonnement sur ces fonctions. Soit la fonction *fact* qui calcule le factoriel d'un nombre naturel n :

```
Fixpoint fact (n:nat) : nat :=
  match n with
    | O => 1
    | S n1 => S n * fact n1
        end.
```

Cette fonction va être décrite par la relation inductive *pfact* qui reliera deux nombres naturels comme ceci :

```
Inductive pfact: nat->nat->Prop: =
      | fact_0: pfact 0 1
      | fact_s: forall (n f:nat), pfact n f->pfact (S n) ((S n) *f).
```

L'idée est la suivante : pour représenter une fonction *f* à *k* arguments, on introduira la propriété inductive à *k+1* arguments qui met en relation les *k* valeurs en entrée avec une valeur en sortie. Ensuite, on donnera à cette propriété une collection de constructeurs couvrant tous les cas apparaissant dans la fonction. On déduira ensuite pour chacun des ces cas, la forme des données en entrée. Ces données en entrée apparaîtront dans la conclusion du constructeur. On détermine ensuite les conditions de chaque cas et on les met comme hypothèses au constructeur et on remplace les appels récursifs de la fonction par des références à la propriété inductive. Finalement, on ajoute les quantifications universelles.

4. Manipulation des preuves

Dans cette partie, nous abordons les techniques de raisonnement en Coq, en commençant par la syntaxe de présentation des théorèmes [03] :

Theorem < *nom_du_théorème* > : < *énoncé* >.
Lemma < *nom_du_lemme*> : < *énoncé* >.

Cet énoncé est appelé le but initial, d'une manière générale un but est la donnée d'une formule à prouver sous certaines hypothèses, celles-ci forme le contexte local d'hypothèses du but. Dans le cas d'un but initial, le contexte initial est vide. Le développement d'une preuve se fait d'une manière interactive par l'usage de commandes appelées tactiques.

4.1 La tactique

Un but est une proposition que l'on souhaite démontrer. Une tactique [03] est une commande en langage Coq qui, appliqué à un sous but, soit réussit en produisant les informations suivantes:

- le résout, c'est à dire terminer la preuve du sous but courant et affiche le nouveau sous but à prouver (s'il y en a, sinon la proposition est prouvée);
- le transforme en un autre sous but avec un contexte différent;
- le transforme en plusieurs sous buts, chacun avec son contexte,
- soit échoue, en laissant le sous but inchangé.

4.2 Quelques tactiques de base

Nous présentons ici quelques tactiques de base ([01], [02], [03], [04]):

- ***Assumption:*** Cette tactique est utilisée pour résoudre un sous but *B* lorsque le contexte courant contient une déclaration de la forme *(v:B)*. Autrement dit, lorsque la propriété qu'on veut prouver fait déjà l'objet d'une hypothèse.

- ***Intro:*** Permet d'introduire des hypothèses et des variables quantifiées dans le contexte courant. Si le but courant est un produit dépendant (*forall x:A, B)* alors le nouveau but sera *B* et la variable x de type A sera poussée dans le contexte local, et si le but courant est un produit non dépendant *A->B*, une nouvelle hypothèse *Hi:A* est introduite dans le contexte et le but courant devient *B*.

- ***Apply term:*** Cette tactique essaie de faire correspondre le but courant avec la conclusion du type de *term* (un terme dans le contexte local). Si elle réussit, elle retourne autant de sous buts qu'il y a de prémisses dans *term.*

- ***Genaralize term:*** Elle permet d'ajouter au but courant une quantification universelle. Le terme *term* doit être une variable définie dans le contexte courant. Si le but est de la forme *(p x)* et que *x* est un objet défini de type *T,* alors, *Generalize x* transforme le but en*: forall x: t, (P x)*.
- ***Absurd term:*** Cette tactique permet de déduire une contradiction (term est de type Prop) et génère deux sous buts: *term* et *~term.* En général, *l*'un de ces deux énoncés est une des hypothèses du but courant. Elle est utile dans les preuves par cas ou certains cas sont impossibles.
- ***Induction term:*** Elle permet de faire une preuve par induction. L'argument *term* doit être une constante inductive. Cette tactique génère un sous but pour chaque constructeur du type inductif concerné et remplace chaque occurrence de *term* dans la conclusion et les hypothèse du but en rajoutant au contexte local des hypothèses d'induction.
- ***Unfold ident:*** Cette tactique s'applique si l'identificateur *ident* apparaît dans le but et est défini. Ainsi toutes les occurrences de *ident* seront remplacées par sa définition dans le but courant.
- ***Split:*** Permet de passer d'un but de la forme *A /\ B* à deux buts *A* et *B.*
- ***Les tactiques Right et Left:*** La tactique *Right* permet de transformer un but de la forme *A \/ B* en un but *B* et la tactique *Left* transforme le même but en un but *A.*
- ***auto:*** Cette tactique essaie d'appliquer autant de fois que possible une base de données (Hint) de théorèmes précédemment prouvés. En d'autres termes, auto profite des théorèmes déjà présents dans la mémoire du système Coq.

Voici des exemples sur l'utilisation des quelques tactiques de base :

Exemple 1 *(intro, left, right)*

```
Coq < Lemma exemple1: forall b:bool, b=true \/ b=false.
1 subgoal
  ============================
   forall b : bool, b = true \/ b = false

exemple1 < intro.
1 subgoal

  b : bool
  ============================
   b = true \/ b = false

exemple1 < elim b.
2 subgoals

  b : bool
  ============================
   true = true \/ true = false
subgoal 2 is:
 false = true \/ false = false
exemple1 < left.
2 subgoals
  b : bool
  ============================
```

```
   true = true
subgoal 2 is:
 false = true \/ false = false
exemple1 < trivial.
1 subgoal

  b : bool
  ============================
   false = true \/ false = false

exemple1 < right.
1 subgoal

  b : bool
  ============================
   false = false
exemple1 < trivial.
Proof completed.
```

Exemple 2 *(apply, assumption)*

```
Coq <  Lemma exemple2: forall a b c :Prop, ((a->b)->c)->b->c.
1 subgoal

  ============================
   forall a b c : Prop, ((a -> b) -> c) -> b -> c

exemple2 < intros.
1 subgoal

  a : Prop
  b : Prop
  c : Prop
  H : (a -> b) -> c
  H0 : b
  ============================
 c
exemple2 < apply H.
1 subgoal
  a : Prop
  b : Prop
  c : Prop
  H : (a -> b) -> c
  H0 : b
  ============================
   a -> b
exemple2 < intro.
1 subgoal
  a : Prop
  b : Prop
  c : Prop
  H : (a -> b) -> c
  H0 : b
  H1 : a
  ============================
```

```
 b
exemple2 < assumption.
Proof completed.
```

Exemple 3 *(induction, auto)*

```
Coq < Lemma exemple3 : forall n, n = n - 0.
1 subgoal
  ============================
   forall n : nat, n = n - 0
exemple3< intro.
1 subgoal
  n : nat
  ============================
   n = n - 0

exemple3 < induction n.
2 subgoals

  ============================
   0 = 0 - 0

subgoal 2 is:
 S n = S n - 0
exemple3 < simpl.
2 subgoal
  ============================
   0 = 0

subgoal 2 is:
 S n = S n - 0
exemple3 < auto.
1 subgoal

  n : nat
  IHn : n = n - 0
  ============================
   S n = S n - 0

exemple3 < simpl.
1 subgoal

  n : nat
  IHn : n = n - 0
  ============================
   S n = S n
exemple3 < auto.
Proof completed.
```

4.3 Les tactiques numériques

Le système Coq fournit trois catégories principales de nombres: les nombres naturels, les nombres entiers et les nombres réels. Pour ces trois catégories de nombres, on dispose de relations d'ordre et de tactiques pour décider l'égalité des formules et la satisfiabilité des systèmes d'inéquation ([01], [02], [12]).

La tactique Ring : La tactique *Ring* est bien adaptée pour résoudre des équations polynomiales entre des valeurs apparaissant dans un anneau ou un semi-anneau. Elle permet de démontrer des égalités ou les membres sont des expressions construites avec une fonction d'addition ou une fonction de multiplication et des valeurs $x_1 ... x_n$. Cependant, cette tactique ne marche pas si des fonctions ou des constructeurs sont insérés au milieu des expressions et n'utilise pas les égalités présentes dans le contexte du but courant pour établir son résultat. Il faut s'assurer que les réécritures adéquates ont été effectuées avant de faire appel à cette tactique. L'exemple suivant montre l'utilisation de Ring pour prouver une égalité:

Exemple 5

```
Coq < Theorem exemple_ring: forall x y:Z, (x+y)*(x+y)=x*x+2*x*y+y*y.
1 subgoal

  ============================
   forall x y : Z, (x + y) * (x + y) = x * x + 2 * x * y + y * y

exemple_ring < intros.
1 subgoal

  x : Z
  y : Z
  ============================
   (x + y) * (x + y) = x * x + 2 * x * y + y *
exemple_ring < ring.
Proof completed.
```

La tactique *Omega :* Cette tactique est très puissante pour résoudre les systèmes d'équation et d'inéquation linéaires sur nat et Z. Elle fonctionne en utilisant toutes les informations qu'elle trouve dans le contexte du but courant. Lorsque les inéquations ne sont pas linéaires, *Omega* peut résoudre le but si les expressions linéaires peuvent être reconnu facilement, en revanche, quand deux variables apparaissent multipliées l'une à l'autre, la tactique ne résout pas le but. Exemple :

Exemple 6

```
Coq < Definition square (x:Z):=x*x.
square is defined
Coq <  Theorem exemple_omega: forall x y:Z, 0<=(square x)->3*(square x)<=2*y
->square x<=y.

1 subgoal

  ============================
   forall x y : Z, 0 <= square x -> 3 * square x <= 2 * y -> square x <= y
exemple_omega < intros.
1 subgoal
```

```
  x : Z
  y : Z
  H : 0 <= square x
  H0 : 3 * square x <= 2 * y
  ============================
   square x <= y
exemple_omega < omega.
Proof completed.
```

Dans cet exemple, le terme (*square x*) est considéré comme une boite noire et les inéquations sont considérées comme linéaires.

La tactique *Field :* Cette tactique fournit la même fonctionnalité que la tactique *Ring* mais pour une structure de corps. Elle considère donc également la division. Pour toute simplification concernant les divisions, elle engendre une obligation de preuve supplémentaire pour assurer que le diviseur est non nul. Exemple :

Exemple 7

```
Coq < Theorem exemple_field: forall x y:R, x<>0->y=y*(1/x)*x.
1 subgoal

  ============================
   forall x y : R, x <> 0 -> y = y * (1 / x) *
exemple_field < intros.
1 subgoal

  x : R
  y : R
  H : x <> 0
  ============================
   y = y * (1 / x) * x

exemple_field < field.
1 subgoal

  x : R
  y : R
  H : x <> 0
  ============================
   x <> 0
exemple_field < exact H.
Proof completed.
exemple_field < Qed.
```

La tactique *Fourier :* La tactique *Fourier* fournit la même fonctionnalité que *Omega* mais pour les nombres réels. Les inéquations considérées doivent se ramener à des inéquations linéaires à coefficients rationnels.

Exemple 8

```
Coq < Theorem exemple_fourier :  forall x y:R, (x < y)%R -> (y + 1 >= x - 1)%R.
exemple_fourier  < intros; fourier.
Proof completed.
```

5. Formalisation des réels

La bibliothèque standard de Coq fournit une librairie de nombres réels formalisés selon une axiomatisation classique [12] : des axiomes de bases ont d'abord été définis (par exemple le commutativité et l'associativité de l'addition sur les réels), ensuite, un ensemble de propriétés élémentaires a été prouvé et des fonctions de base ont été définies (max, min, valeur absolue…).

Sur la base de ces axiomes, propriétés et fonctions, plusieurs développements ont été réalisés comme la formalisation de l'analyse (les suites, les séries numériques…). Des tactiques ont été également développées spécialement pour l'ensemble des réels (field, DiscR, SplitRmult…).

Cette formalisation a permis d'effectuer des preuves formelles pour des cas non triviaux issus de domaines différents:

- La démonstration du théorème des trois intervalles qui est un problème mathématique faisant intervenir des notions d'analyse, propriétés numériques et géométrie [12] ;
- La démonstration d'une partie du logiciel *O*dyssée : un logiciel de différentiation automatique de programme FORTRAN-77 ([12], [16]).

6. Preuves des programmes dans Coq

Etant basé sur le CCI, le système Coq a été d'abord utilisé pour prouver des programmes fonctionnels [03] [15]. L'utilisation de ce système pour prouver des programmes impératifs remonte à 1999 avec le développement par J-C FILLIATRE d'une méthode, dans Coq, qui porte le nom *Correctness* [08].

L'implantation de cet méthode se présente sous la forme d'une tactique qui prend en entrée un programme écrit dans un langage dédié proche de ML et pouvant contenir des constructions impératives et des annotations dans le style de la logique de Hoare, selon la syntaxe suivante:

Correctness *nom_programme-annoté.*

Cette commande engendre alors une série d'énoncés qui correspondent aux propriétés logiques à vérifier pour s'assurer de la correction du programme. Ces énoncés peuvent alors être démontrés par l'utilisateur dans le système Coq.

Par la suite, *Correctness* a été dissociée du système Coq, pour devenir un outil, plus élaboré, dédié à la vérification de programmes impératifs : l'outil Why[05].

CHAPITRE 4

Les outils Caduceus et Why

Chapitre 4

Les outils Caduceus et Why

1. Introduction

Dans le but d'en effectuer la vérification, un programme C annoté de sa spécification est d'abord soumis à l'outil Caduceus qui génère un programme écrit dans le langage de l'outil Why (WL: Why Language). L'outil Why est un VCG (Verification Condition Generator): à partir du programme écrit en WL, il génère des obligations de preuves qui sont des formules logiques dont la validité implique la correction du programme vis-à-vis sa spécification. Ces formules logiques peuvent alors être prouvées dans le système Coq (figure 4.1).

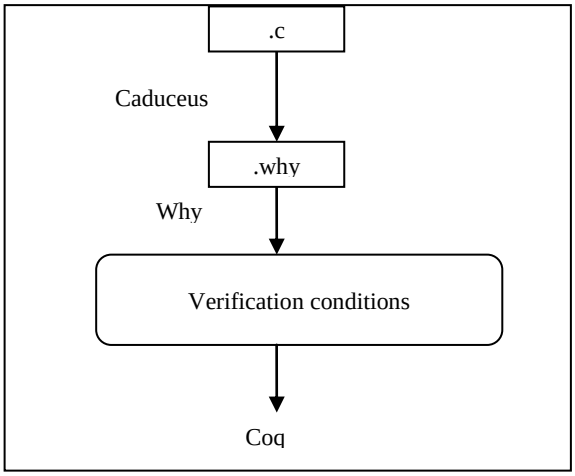

Figure4.1: Combinaison des outils Caduceus, Why et Coq

2. L'outil Caduceus

Caduceus ([06], [09]) est un outil de vérification pour les programmes C, et plus précisément de programmes répondant à la norme ANSI-C. Il permet à la fois de vérifier l'absence de *menaces* (accès à l'adresse d'un pointeur nul, appelé déréferencement, ou l'accès en dehors des bornes d'un tableau) et de prouver des propriétés fonctionnelles d'un programme, spécifiées sous la forme d'annotations insérées dans le code source.

2.1 Annotations et spécifications

Une annotation [09] de programme est un énoncé placé au cœur même du programme dont on cherche à vérifier le comportement. Les annotations sont utilisées pour spécifier les propriétés fonctionnelles de chaque fonction du programme dans un langage de spécification donné. D'un point de vue pratique, les annotations sont insérées dans le code source C sous la forme de commentaires spéciaux : /*@...@*/. Les annotations sont écrites dans un langage de spécification de premier ordre qui étend la syntaxe du langage C (les expressions sans effets de bord) avec de nouveaux constructeurs, prédicats et mots-clé. La spécification d'un programme consiste à décrire, sous forme d'annotations, le comportement attendu de chaque fonction du programme, c'est-à-dire, les propriétés qui doivent être vérifiées par les variables intervenant dans l'exécution de la fonction.

La spécification d'une fonction [09] comporte sa précondition et sa post-condition qui sont des assertions logiques portant sur les variables et éventuellement sur le résultat de la fonction :

a- la précondition décrit les propriétés qui doivent être vérifiées par les variables au moment ou la fonction est appelée. Elle est introduite dans l'annotation par le mot clé : *requires.*

b- La postcondition décrit les propriétés vérifiées par les variables après l'exécution de la fonction (sous réserve que les propriétés décrites dans la précondition soient vérifiées). Elle est introduite dans l'annotation par le mot clé : *ensures.*

c- Une clause qui spécifie toutes les variables qui peuvent être modifiées par la fonction (appelées valeurs gauches). Cette clause est introduite dans l'annotation par le mot clé : *assigns.*

En plus des annotations qui décrivent la spécification des fonctions, d'autres doivent être ajoutées pour spécifier les boucles qu'un programme contient : ce sont les annotations de boucles [09]. Elles sont nécessaires pour pouvoir prouver les obligations, qui seront générées par l'outil Why, et contiennent :

- Le variant de boucle qui permet d'assurer la terminaison de la boucle en indiquant quelle est l'expression qui décroît strictement à chaque tour de boucle. Il est introduit dans l'annotation par le mot clé : *variant.*
- L'invariant de boucle qui exprime la propriété qui est vérifiée à chaque tour de boucle par les variables intervenant dans le corps de la boucle. (il est vérifié à l'entrée de la boucle, à chaque itération et à la sortie). Il est introduit dans l'annotation par le mot clé : *invariant.*
- Une clause qui indique l'ensemble des expressions qui sont modifiées par la boucle, introduite par le mot clé : *loop_assigns.*

Lorsqu'une précondition contient une propriété, portant sur une ou plusieurs variables globales, qui est vraie dans l'ensemble du programme, cette propriété peut être décrite sous la forme d'un *invariant global* [09]. Elle sera alors ajoutée aux préconditions de toutes les fonctions manipulant ces variables et aux post-conditions modifiant les dites variables. De plus, l'outil Caduceus nous permet (en plus des pré/post-conditions d'une fonction) d'écrire des annotations à des points choisis dans le corps d'une fonction, sauf dans le corps d'une boucle [09]. Ceci se fait quand on a besoin de prouver une propriété intermédiaire qui sera utilisée pour prouver la postcondition par exemple.

La figure 4.2 montre l'annotation d'un petit programme C qui calcule le modulo (le reste de la division de l'entier x par l'entier y) [05]:

```
/*@ requires x>=0 && y>0
@ ensures 0 <= \result < y &&
@ \exists int d; x== d * y + \result
@*/
int math_mod (int x, int y) {
/*@ invariant 0 <= x && \exists int d; \old(x) == d * y + x
@ variant x
@*/
 while (x >= y) x -= y;
return x;}
```

Figure 4.2: La fonction Modulo annotée de sa spécification

- Le mot-clé *requires* introduit la précondition: (x>=0 et y>0)
- Le mot-clé *ensures* introduit la postcondition de la fonction: le résultat retourné par la fonction et dénoté par *result* est tel que : (0 <= \result < y) et il existe un entier d (@ *exists int d)* tel que x est égale à la somme du produit (d*y) et du résultat retourné.
- Le variant de la boucle while est la valeur x
- L'invariant de cette boucle exprime que nous avons toujours (x >= 0) et à n'importe quelle étape de la boucle, il existe un entier d tel que la valeur initiale de x, dénotée par *old(x)* et qui veut dire : la valeur de x à l'appel de la fonction est la somme du produit (d * y) et de la valeur de x actuelle.

2.2 La traduction Caduceus

2.2.1 Le modèle mémoire dans Caduceus [09]

Caduceus est un outil construit sur la base de l'outil Why. Il traduit les programmes C dans le langage d'entrée de Why. Dans ce langage, les types sont uniquement des types fonctionnels purs et des références sur ces types, sans aucune notion de tableaux, structures ou pointeurs. Par conséquent, la traduction effectuée par Caduceus nécessite un modèle mémoire du langage C, de façon à représenter les structures du langage C et l'état courant d'un programme C donné.

Pour le modèle mémoire, Caduceus suit une approche de Burstall-Bornat où la mémoire est divisée en emplacements disjoints, ou adresses de base, correspondant aux champs de structures et aux blocs alloués par des pointeurs dans le programme. Les champs d'une structure sont considérés comme alloués dans des emplacements disjoints. De même, deux pointeurs appartiennent au même emplacement s'ils pointent dans le même bloc alloué, ce qui signifie qu'il y a eu une affectation entre les deux pointeurs. Une arithmétique de pointeurs est possible à l'intérieur de l'emplacement, mais le pointeur ne pourra pointer vers un autre emplacement mémoire que par une affectation à un autre pointeur.

Cette séparation assure « gratuitement » qu'une modification dans un emplacement mémoire n'affecte pas les autres emplacements. Si deux pointeurs appartiennent à des emplacements disjoints, il ne sera pas nécessaire de prouver que la modification de l'un ne modifie pas l'autre.

2.2.1.1 Le type « pointer »

Dans ce modèle mémoire, il y a deux types de base: les types numériques (les entiers et les flottants) et un nouveau type *pointer*. Une valeur *p* de type *pointer* est soit le pointeur nul, soit une paire *(base_addr (p), offset (p))* composée de l'adresse de base de l'emplacement mémoire ou le pointeur a été alloué et l'index ou pointe *p* à l'intérieur de ce bloc. Les notions de tableaux C et de pointeurs C sont confondues dans le modèle et sont toutes deux associées au type *pointer*.

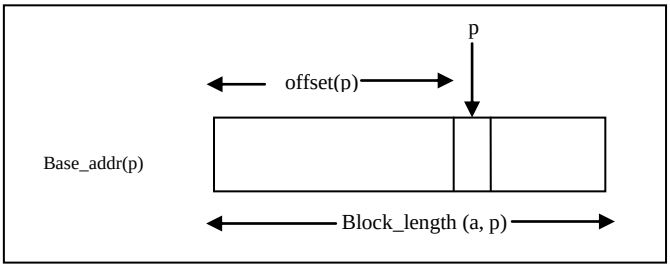

Figure 4.3: Le type pointer

2.2.1.2 Les états mémoire

Un état mémoire d'un programme C est représenté par un ensemble de variables globales Why, englobant, de façon structurée, tous les emplacements mémoire du programme. En particulier, une variable *intP* sera utilisée pour représenter l'ensemble des segments mémoires contenant un pointeur d'entiers à un état donné du programme. Cette variable peut donc être vue comme associant à chaque adresse de base d'un pointeur, le segment mémoire ou a été alloué le pointeur. Une variable supplémentaire, notée *alloc* représente la table d'allocation à un instant donné. Cette variable indique quelles sont les adresses qui sont allouées et quelle est la taille du bloc *(block_length)* sur lequel elles pointent. Par conséquent, un état donné du programme est généralement représenté dans le modèle mémoire de Caduceus comme indiqué dans la figure 4.4.

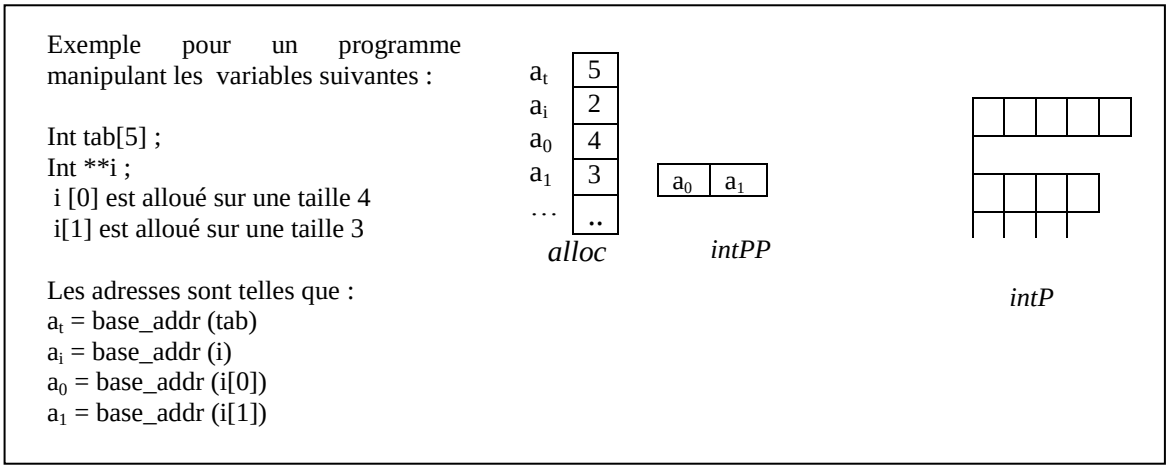

Figure 4.4 : Représentation d'un état mémoire

2.2.1.3 La théorie associée au modèle

Le modèle mémoire défini par Caduceus est associé à une théorie permettant de manipuler et de raisonner sur les composants de ce modèle. Nous y trouvons:

1- ***Des prédicats de validité*** qui permettent de définir la validité des pointers, à savoir la condition pour qu'ils puissent être déréférencés (pour que leurs adresses soient accédées). Un pointeur peut être déréférencé s'il n'est pas le pointeur nul et s'il pointe dans un segment alloué. La validité d'un pointer p, la validité d'un indice i est d'un ensemble d'indices allant de i à j sont respectivement exprimées par :
valid (alloc,p) $\equiv p \neq null \wedge 0 \leq offset\ (p) < block\_length\ (alloc\ ,p)$
valid_index (alloc,p,i) $\equiv p \neq null \wedge 0 \leq offset\ (p) + i < block\_length\ (alloc,p)$
valid_range(alloc, p,i,j) $\equiv 0 \leq offset\ (p) + i \wedge i \leq j \wedge offset\ (p)+j < block\_length\ (alloc,p)$

2- ***Des fonctions d'accès et de modification*** qui permettent la gestion des variables Why et la modélisation des opérations du langage C. Nous citons par exemple: une fonction d'accès *acc (m, p)* retourne la valeur contenue à l'index offset (p) dans le tableau associé à *base_addr (p)* par l'état mémoire m. Une fonction de mise à jour *upd (m, p, v)* retourne, quant à elle, un nouvel état m1, ou la valeur « pointée » par p devient v. Une fonction d'arithmétique de pointeur *shift (p, i)* permet de représenter l'expression p+i ou p[i] du langage C. Ces fonctions sont définies de façon axiomatique dans Why. Voici quelques axiomes régissant l'utilisation de ces fonctions:
acc (upd (m,i,v),i) = v
$j \neq k$ *-> acc (upd (m,i,v) , k) = acc (m, k)*
shift (shift (p,i),j) = shift (p, i+j).

3- ***La clause assigns*** Dans les annotations Caduceus, la clause *assigns* (resp. *loop_assigns*) permet d'indiquer l'ensemble des expressions qui sont modifiées par la fonction (la boucle). Cette clause est modélisée par le prédicat suivant:
assigns (alloc, m1, m1, loc) $\equiv \forall$ *p: pointer, valid (alloc,p)* $\wedge$ *unchanged (p,loc) ->acc (m2,p)=acc(m1,p)*
Ceci indique que seules les variables contenues dans la location (ensemble de pointers) *loc* ont été modifiées entre les états mémoires m1 et m2. Le prédicat *unchanged* est défini par:
$\forall$ p1:pointer, $\forall$ p2:pointer, p1$\neq$ p2 -> unchanged (p1, (pointer_loc p2)).

2.2.2 Le calcul d'effets dans Caduceus

Une fois les variables représentant les états mémoires déterminés, l'étape qui suit consiste à calculer les effets de chaque fonction [09]. En effet, les spécifications des programmes dans le langage d'entrée de Why doivent expliciter les effets de bords et les variables lues par les fonctions. La spécification d'une fonction Why annotée a la forme suivante:

Pararameter f-parameter :
A :t1-> ...->an :tn -> {Pre} T reads v_1*,...,* V_m *writes* w_1*,...,* w_p *{Post}.*

Un effet est une pair de l'ensemble des variables Why respectivement lues et modifiées, correspondant aux clauses *reads* et *writes.* Les effets sont calculés par une analyse statique du programme [11].Nous reviendront sur ce point lors de la présentation de l'outil Why.

2.2.3 La traduction de codes C dans Caduceus

Une fois les variables qui modélisent les états mémoires du programme déterminées et les effets de chaque fonction calculés, Caduceus traduit la sémantique des constructions C en instructions Why, par affectations de ces variables globales et l'ajout d'annotations (comme l'ajout automatique de prédicats de validité). Notons que les annotations du code C sont transformées en prédicats Why sur les variables Why. La traduction du code C est réalisée par quatre interprétations mutuellement récursives [06]:

- [e] : L'interprétation des expressions C ;
- $[e]_b$: L'interprétation d'expressions C comme des expressions Why booléennes ;
- $[e]_l$: L'interprétation des valeurs gauches du programme C : ceux sont des expressions qui apparaissent dans la partie gauche d'une affectation ;
- $[s]_s$: L'interprétation des constructions C.

La figure 4.5 présente quelques règles de traduction concernant ces quatre interprétations.

[x] = ! x
*[*e] = access m p, si* $[*e]_l$ *= (m, p)*
[e1 = e2] = let v1 = p1 in let v2 = [e2] in updte m v1 v2; v2, si $[e1]_l$ *= (m, p1)*
*[e1 op e2] = [e1] op [e2], avec op dans { +, -, *, /, %, &, ^, , |}*
[e1 op e2] = if $[e1 \text{ op } e2]_b$ *then 1 else 0, avec op dans { ==, !=, >, >=, <, <=, &&, ||}*
$[e1 \text{ op } e2]_b$ *= let v1 = [e1] in let v2 = [e2] in v1 op v2 avec op dans { ==, !=, >, >=, <, <=}*
$[e1 \&\& e2]_b$ *= if* $[e1]_b$ *then* $[e2]_b$ *else false*
$[e1; e2]_s$ *= [e1] ; [e2]*
…

Figure 4.5 : Quelques règles de traduction de codes C

3. L'outil Why

3.1 Introduction

L'outil Why est un générateur de conditions de vérification. Il prend en entrée un programme annoté et produit des obligations de preuves dont la validité assure la correction du programme vis-à-vis sa spécification (les annotations). Les obligations de preuves sont des formules en logique du premier ordre exprimables dans la syntaxe de différents systèmes de preuves existant, aussi bien des assistants de preuves comme Coq et PVS, que les démonstrateurs automatiques comme Simplify ou haRVey.

L'outil Why n'est pas limité à un langage particulier. Il fournit un langage (WL: Why Language) vers lequel les langages existant peuvent être traduits. Il permet donc de prouver des programmes C avec l'outil Caduceus et des programmes JAVA à l'aide de l'outil KRAKATOA. La figure 4.6 montre le fait que Why est un outil multi-langages et multi-prouveurs

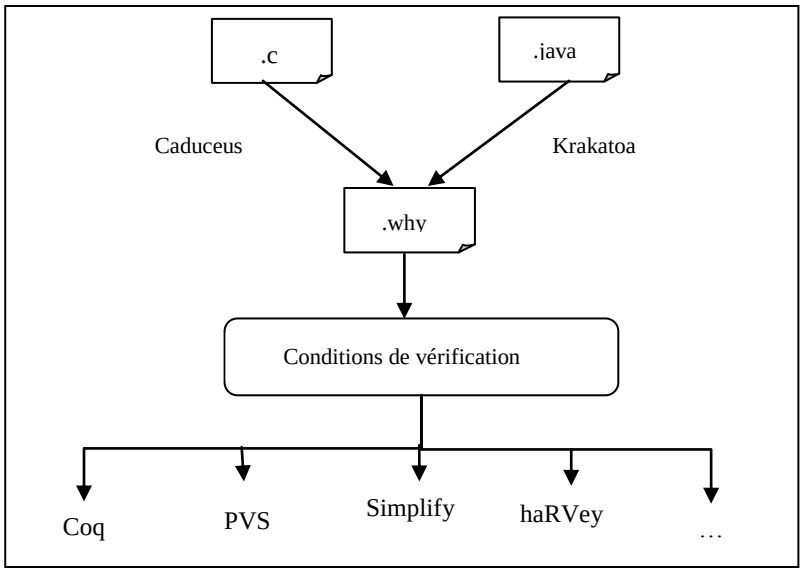

Figure 4.6 : Caractéristiques de l'outil Why

3.2 Le langage Why

C'est un langage fonctionnel (proche du langage Caml) auquel sont ajoutés quelques traits impératifs tel que [47]:

Les types de base sont : les entiers : int, les flottants : float, les booléens : bool et le type unit habité par la constante void.

Le noyau fonctionnel : Il est constitué des expressions suivantes :

- les constantes entières, flottantes, booléennes et void ;
- les variables ;
- l'application, notée sous forme currifiée *(e e_1 …e_n)* ;
- les opérations primitives (unaires et binaires) ;
- les définitions locales : *let v = e_1 in e_2 ;*
- la conditionnelle : *if e_1 then e_2 else e_3* ;
- l'abstraction notée *: fun (x_1 : t_1)… (x_n : tn) ->e.*

Les traits impératifs : sont introduits par :

- les définitions des références locales *: let v = ref e_1 in e_2* ;
- la déréférenciation *! v ;*
- l'affectation *v :=e ;*
- la séquence *e_1 ;… ; e_2 ;*
- la boucle : *while e_1 do e_2 done;*

L'outil Why permet à l'utilisateur la déclaration de modèles logiques (types, fonctions, prédicats et des axiomes) qui peuvent être utilisés soit dans le programme soit dans l'annotation. Il permet par exemple de définir de manière axiomatique des structures de données complexes telles que les tableaux.

3.3 La méthode Why

La méthode Why repose sur la traduction fonctionnelle des programmes impératifs, la construction d'un terme de preuve et le calcul de plus faible précondition [08].

3.3.1 La traduction fonctionnelle

Un programme impératif peut être traduit en un programme fonctionnel par ajout de paramètres. Cela nécessite une analyse statique des effets du programme ainsi que l'utilisation de la notion de monade [08].

L'analyse des effets est la première étape réalisée par la traduction fonctionnelle Why [08]. Les effets sont les comportements à l'exécution qui ne font pas partie du noyau fonctionnel. Pour le langage Why, tel qu'il est présenté au paragraphe (3.2), il s'agit surtout des effets de bord sur les références : création, lecture et modification.

Soit le fragment de code Ocaml suivant qui ajoute à la référence ***s*** la somme des ***i*** premiers entiers. Sa traduction purement fonctionnelle est un programme prenant en argument les valeurs des variables ***i*** et ***s*** et retournant leurs nouvelles valeurs. Il s'agit du code de la figure 4.7.

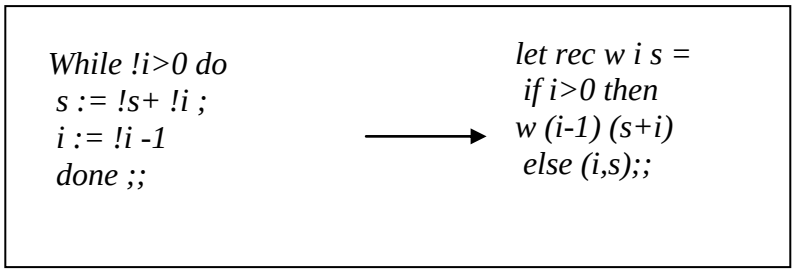

Figure 4.7 : Traduction fonctionnelle d'un code impératif

Dans cet exemple, les effets de la boucle sont un accès en lecture et écriture des références i et s. C'est cette information qui permet de déduire la traduction fonctionnelle. Si par exemple, le programme ci-dessus ne modifiait pas la valeur de s mais, se contentait de la lire, alors la traduction fonctionnelle aurait été une fonction prenant en argument les valeurs de i et s mais retournant seulement la valeur de i.

L'idée est qu'une expression du programme accédant au contenu d'une référence *r* devient un terme fonctionnel prenant en argument la valeur de *r*. Une expression modifiant la référence r devient un terme fonctionnel retournant, éventuellement parmi d'autres valeurs, la nouvelle valeur de *r*. Un effet va donc être composé de deux ensemble de références : les références potentiellement lues par l'expression du programme et les références potentiellement modifiées par celle-ci.

La méthode Why définit pour WL un système de type avec effets ainsi qu'un algorithme de typage avec inférence des effets. La notion de type y est enrichie avec les effets. Prenons l'exemple de la fonction *créditer* (en WL) suivante [47] :

parameter m : int ref
let créditer = fun (s :int) -> {s>=0} m :=!m + s{m = s+@m}

Où: *{s>=0}* est la précondition, *{m = s+@m}* est la postcondition et *@m* désigne la valeur initiale de la référence *m.*

En tant que programme « Ocaml », on sait que le type de cette fonction est *int->unit.* Ce qui est fait en plus, est le calcul des effets, pour déterminer dans cet exemple, que la référence *m* est à la fois lue et écrite. Un type avec effets est donc un triplet (type, variables lues, variables écrites). Pour la fonction *créditer*, on aura donc le type : *(int-> unit, {m}, {m}).*

Le concept de ***monade***, utilisé par la traduction fonctionnelle Why, permet de simuler des traits impératifs (les exceptions, les entrées/sorties...) dans un cadre purement fonctionnel. Les monades ont été introduites par P.WADLER dans [10] où il montre leur utilisation dans le langage purement fonctionnel Haskell. Cette technique consiste à remplacer une fonction qui a le type ***a->b*** (une fonction prenant un argument de type ***a*** et retournant un résultat de type ***b***) par une fonction de type ***a->M b*** c'est à dire une fonction qui prends un argument de type ***a*** et retourne un résultat de type ***b,*** avec des effets « capturés » par ***M.***

Prenons un exemple [10], en Haskell, pour expliquer les monades. Il s'agit d'écrire une fonction *eval* qui prend un argument de type *term* et retourne sa valeur entière.

Le type de donnée *term* est donné par la définition suivante :

Data Term = Cons Int | Div Term Term

Un *Term,* est soit une constante entière (le constructeurs *Cons*), soit une division d'un *Term* sur un *Term* (le constructeur *Div*). La fonction d'évaluation s'écrit alors comme ceci :

eval :: Term -> Int (Une fonction eval de type Term -> Int)
eval (Cons a) = a
eval (Div t u) = eval t / eval u (« / » pour la division entière)

Maintenant, si on veut modifier cette fonction de telle sorte qu'elle lève une exception en cas de division par zéro, on utilise la technique suivante :

Data M a = Raise Exception \ Return a

Cette ligne introduit un nouveau type de donnée (la monade *M*) pour représenter un calcul qui peut lever une exception. Un calcul est soit une valeur de type *a*, soit une exception.

Type Exception = String (Introduction d'un nouveau nom pour le type existant String)

La fonction *eval* s'écrit alors comme suit :

```
eval :: Term -> M Int          (le type de la fonction devient Term ->M Int)
eval (Con a) = Return a
eval (Div t u) = Case eval t of        (évaluation selon la valeur retournée par eval t)
          Raise e -> Raise e
         Return a -> Case eval u of  (évaluation selon la valeur retournée par eval u)
                    Raise e -> Raise e
                    Return b -> if b= 0 then Raise "Division par zero"
                                else Return (a/b)
```

A chaque appel de l'évaluateur, la forme du résultat est vérifiée : si c'est une exception, elle est propagée est levé, sinon, le calcul est effectué.

La monade *M* peut être modifiée pour englober d'autres traits impératifs comme les entrées/sorties et les états [10]. L'outil Why suit la même idée en donnant une monade généralisée grâce au système de type avec effets qui fournit des renseignements riches sur les références accédées ou modifiées individuellement. La monade est dite généralisée dans le sens qu'elle traite tous les cas sur les références et les exceptions.

3.3.2 Méthodologie de preuves

La traduction fonctionnelle constitue une étape vers la génération des obligations de preuve pour établir la correction et la terminaison des programmes. Pour cela, Why introduit une notion de type annoté avant la génération des obligations de preuves.

La méthode Why assimile spécification et type [08]: la spécification est considérée comme une extension du système du type avec effets en y incorporant des annotations : des pré/postconditions pour ***chaque*** expression du programme. C'est cette annotation qui va permettre de prouver des programmes, autrement exprimé :

Monades+ types avec effets → traduction fonctionnelle
Types avec effets annoté → preuve des programmes

Par exemple, pour la fonction *créditer* précédente, on obtient le type avec effets annoté suivant : *int -> {s>=0} (unit, {m}, {m}) {m = s+@m}*

La méthode Why introduit ensuite une interprétation des types annotée dans la théorie des types, plus spécialement dans le calcul des constructions inductives. De façon générale, un type annoté *T= {P} (t, R, W) {Q}* tel que *R* est l'ensemble des variables lues et *W* celui des variables écrites est interprété en un type CCI : $\forall$ *X, P(X)* $\rightarrow$ $\exists$ *Y, Q(X, Y)*, où :

La notation *P(X)* désigne la formule *P* où les occurrences des variables lues sont substituées par les *X* (*X* est un vecteur de valeurs), et la notation *Q(X, Y)* désigne la formule *Q* où pour chaque variable *v* de *W*, les occurrences de *v* sont remplacées par le y correspondant, les *@v* par le *x* correspondant. Pour chaque variable *v* de *R* qui n'est pas dans *W*, chaque occurrence de *v* est remplacée par le *x* correspondant et *r* dénote le résultat. Il s'agit là d'exprimer le fait que les *x* désignent les anciennes valeurs modifiables et les *y* désignent les nouvelles valeurs.

L'étape qui vient après la définition de la notion de spécification consiste à ***engendrer un ensemble d'obligations de preuves*** à partir d'une spécification et d'un programme annoté. Cette méthodologie est basée sur le calcul des constructions inductives (CCI) où on retrouve une distinction entre une partie informatives qui contient les types de données et la partie logique qui contient les prédicats. Le mécanise de l'extraction exploite le caractère constructif des preuves dans le CCI pour extraire à partir d'une preuve de la proposition $\forall\ x, P(x) \rightarrow \exists y, Q(x, y)$, telle que *P* et *Q* sont des prédicats et *x* et *y* de type informatifs, une fonction *f* calculant *y* en fonction de *x, et* qui possède la propriété suivante: $\forall\ x, P(x) \rightarrow Q(x, f(x))$.

L'idée de base de la méthodologie Why est alors la suivante [08]: à partir d'un programme *e* est d'une spécification *k* (qui est un type avec effets annoté), construire une preuve $\hat{e}$ de la proposition qui est l'interprétation dans le CCI de *k* (et qui a donc la forme $\forall\ x, P(x) \rightarrow \exists y, Q(x,y)$) de telle sorte que le terme de preuve $\hat{e}$ possède un contenu extrait égale à la traduction fonctionnelle e_f de *e* et qui a la propriété : $\forall\ x, P(x) \rightarrow Q(x, e_f\ x)$. C'est-à-dire : pour toute entrée *x* vérifiant *P*, alors le résultat du programme (e_f x) satisfait la postcondition *Q* , et c'est exactement ce que l'on entend par correction du programme *e* vis-à-vis sa spécification *k*.

La construction intégrale du terme de preuve n'est pas possible automatiquement, ce terme sera donc incomplet. Ses *« morceaux manquant »*, sont les obligations de preuves laissées à la charge de l'utilisateur. Seule la partie informative peut être construite avec l'utilisation des monades et des types avec effets. Prenons un exemple [07] :

Soit le triplet de Hoare : $\{x>0 \wedge y>1\}\ x:=x+1;\ y := y*x\{y> @y\}$

La partie calculatoire consiste à interpréter les deux affectations comme le calcul de deux valeurs finales x_1 et y_1 à partir de valeurs initiales x_0 et y_0. La partie logique consiste en l'hypothèse $x_0>0 \wedge y_0>1$ et la preuve de $y_1>y_0$ (qui constitue la condition de vérification). La traduction fonctionnelle est la suivante :

$$\lambda\ x_0, y_0.\ \lambda\ h : x_0>0 \wedge y_0>1.$$
$$\text{let } x_1 = x_0+1 \text{ in}$$
$$\text{let } y_1 = y_0*x_1 \text{ in}$$
$$(x_1, y_1, \square : y_1> y_0)$$

Le symbole □ représente l'obligation de preuve dont l'énoncé est : $x_0>0 \wedge y_0>1 \Rightarrow y_0*(x_0+1)>y_0$. Notons ici que la traduction fonctionnelle est donnée en λ-calcul, qui est la base de tout formalisme fonctionnel.

3.3.3 Calcul de la plus faible précondition

Avec l'inférence de type avec effets, Why associe à chaque sous expression d'un programme un type avec effets. L'étape suivante consiste à leurs associer une précondition et une postcondition : de deux choses l'une : soit l'expression considérée est explicitement annotée par l'utilisateur, soit on lui détermine une annotation par le calcul de la plus faible précondition [07]. Ce calcul consiste à modéliser le programme sous la forme d'un transformateur de prédicats, c'est à dire une fonction *WP* qui à toute proposition Q associe la plus faible précondition que doivent vérifier les variables avant l'exécution du programme pour que Q soit vérifiée après l'exécution.

Le calcul de la plus faible précondition se fait en remontant le corps de la fonction, instruction par instruction. Lors de ce calcul, l'outil Why considère l'appel de fonction et les boucles comme des boites noires dont on ne connaît que la spécification. Cela permet un calcul immédiat de la plus faible précondition, sans dérouler la boucle ou le corps de la fonction appelée.

- **L'appel de fonction:** Seule la spécification de la fonction est utilisé par le calcul. Sa précondition doit être vérifiée au moment de l'appel et la post-condition doit impliquer la propriété recherchée.
- **Les boucles:** C'est l'invariant de la boucle qui est utilisé lors du calcul, sous réserve qu'elle vérifie effectivement la propriété exprimée dans l'invariant**.** (Figure 4.8).

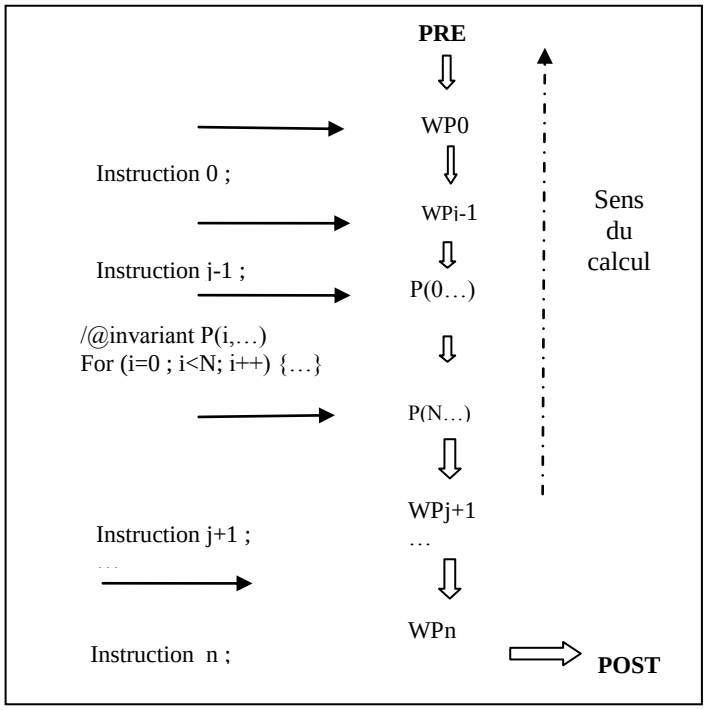

Figure 4.8 : Schéma du calcul de plus faible précondition pour la boucle

- **Les expressions annotées :** une expression annotée {p'} s {q'} d'un programme donné {P} e {Q} n'est pas consultée lors du calcul de la plus faible précondition du programme. Le calcul traite ce genre d'expressions comme suit: *wp ({p'} s {q'}, Q) = p' /\ ∀ result. ∀ w. q' => Q*. tel que w est l'ensemble des variables qui peuvent être modifiées par s.

Voici un extrait des formules de calcul de WP [07] :

wp (t, q) = q[result <- t]
wp (!x, q) = q[result <- x]
wp (x:=e,q) = wp (e, q [result <- void; x<- result])
wp (e1; e2, q) = wp (e1, wp (e2,q))
wp (if e1 then e2 else e3, q) = wp (e1, if result then wp (e2,q) else wp (e3, q))
wp (let x = e1 in e2, q) = wp (e1, wp (e2, q)[x <- result])
…

Figure 4.9 : Quelques règle du calcul WP

Le calcul de la plus faible précondition permet donc d'étendre le système de types en y incorporant les annotations : on obtient pour une expression d'un programme *un type avec effets annoté* (qui constitue en fait la spécification de l'expression) et qui est assez riche pour permettre d'obtenir les obligations de preuves.

4. Exemple d'utilisation

L'exemple suivant illustre l'utilisation des outils Caduceus et Why afin de vérifier qu'un programme satisfait sa spécification. Soit la fonction C *som_tab* qui prend en argument un tableau *t* et un entier *n* retourne la somme des *n* premier élément du tableau.

```
/*@Logic int somtab (int* t, int n) reads t[..] @*/
/*@ requires n>0
@ \valid_range (t, 0, n-1)
@ensures \result == somtab  (t, (n-1))
@*/
int som_tab (int t[20], int n )
 {
int i,s;
s=0;
/*@invariant (0<=i<=n) && (s == somtab  (t, (i-1)))
@variant n-i
@*/
for(i=0;i<n;i++) {
s=s+t[i];}
return s; }
```

Figure 4.10 : La fonction som_tab annotée

La figure 4.10 montre la fonction som_tab annotée de sa spécification, les annotations sont mises en évidence en italique. Il s'agit de :

- La précondition qui indique que les accès au tableau doivent être valide via le prédicat prédéfini *valid_range (t, 0, n-1)* indiquant que tous les accès de *t[0]* à *t[n-1]* sont valides et *(n>0)*
- La poste condition qui exprime que le résultat retourné par la fonction est égal à la valeur calculée par la fonction *somtab*. C'est une fonction logique utilisée ici pour exprimer la somme des éléments du tableau t de *0* à *(n-1)*. Notons que cette fonction est déclarée comme objet logique, sa définition se fera au niveau du système Coq ultérieurement, étant donné que le langage Caduceus ne permet pas de la spécifier.
- L'invariant qui exprime que la valeur de *i* est toujours incluse entre *0* et *n* et qu' à l'entré de la boucle et à une étape *i* de son déroulement, la valeur de *s* est obtenue en sommant les éléments de *t* dont l'indice varie de *0* à *(i-1)* (exprimé par la fonction *somtab*).
- Le variant de cette boucle est *(n-i)*.
- L'écriture *« reads t[..] »* indique les locations mémoire qui sont concernées par la fonction logique *somtab*.

Une fois le code *(somme.c)* correctement annoté, on le soumet à l'outil Caduceus par la commande ***Caduceus somme.c***. Cette commande permet de générer le code Why équivalent à partir duquel les obligations de preuves vont être générées pour le système Coq par la commande suivante : ***make –f somme.makefile coq***.

Cette dernière commande permet de générer les deux fichiers Coq suivants :

- *somme_why.v* : Ce fichier contient les obligations de preuves (lemmes) qu'il faut démontrer. Ils concernent : la validité des accès au tableau, la validité de l'invariant à l'entrée de la boucle, la préservation de l'invariant, la terminaison de la boucle et la postcondition.
- *somme_spec_why.v* : Ce fichier contient des spécifications du programme considéré : si le programme contient des déclarations d'objets logiques, il faut les définir pour pouvoir effectuer les preuves. Pour notre exemple, nous avons déclaré la fonction *somtab*, qui sera définie en Coq (Figure 4.11).

```
    Fixpoint somtab_aux (m:memory Z global) (t:pointer global) (i:Z) (n:nat) { struct n} :Z:=
      match n with
| 0%nat =>(shift t i #m )%Z
| S n1 => (shift t i #m)%Z + (somtab_aux m t (i-1) n1)%Z
end.

Definition somtab (m:memory Z global) ( t: pointer global) (i:Z):= match (i ?=0) with

| Lt => 0
| _ => somtab_aux m t i (Zabs_nat (i))
end.
```

Figure 4.11 : Définition de la fonction somtab dans Coq

Explications

La fonction *somtab* prend trois arguments : un état mémoire *m*, un pointer *t* et un entier *i*. Elle renvoie zéro si elle est appelée avec *i<0* (le résultat de la comparaison de *i* avec *0* dans *« match (i ?=0) with »* est *Lt)* - ce qui permet de prouver l'invariant à l'entrée de la boucle : elle sera appelée avec *i = (0-1)*- sinon, elle appelle la fonction *somtab_aux*.

La fonction somtab_aux est une fonction récursive qui prend quatre arguments : un état mémoire, un pointer, un entier et un nombre naturel *n* qui représente l'argument de la récursivité. Si cet argument est zéro, alors elle retourne la valeur entière *(shift t i #m)* qui signifie : la valeur *t[i]* à l'état *m*, sinon si la forme de *n* est *(S n1)*, elle fait l'appel récursif *: (shift t i #m)%Z + (somtab_aux m t (i-1) n1)%Z* . Notons que la fonction *somtab* appelle *somtab_aux* avec un entier *i* et la valeurs *Zabs_nat (i)* : la valeur naturelle de l'entier *i*, dans le but de fait la récursivité sur les entiers naturels.

CHAPITRE 5

Contribution
Application de la méthode Why
A la certification de programmes
D'algèbre linéaire

Chapitre 5

Contribution
Application de la méthode Why à la certification de programmes d'algèbre linéaire

1. Introduction

1.1 Les bibliothèques numériques

Le calcul scientifique utilise de façon intensive les opérations vectorielles et matricielles et demande une grande quantité de calcul. Les utilisateurs se sont donc trouvé face à la nécessité de trouver des moyens d'obtenir les résultats le plus rapidement possible. De plus, la multiplication des plates-formes de calculs aidant, un autre problème à résoudre a été de faire en sorte que les codes soient transportables entre différentes machines et différents compilateurs [69].

La réponse apportée a été de mettre au point des bibliothèques qui prennent en charge les opérations matricielles. Le but des bibliothèques numériques est le suivant [71] : premièrement, elles servent à offrir à l'utilisateur une interface souple pour résoudre des problèmes communs à diverses applications tout en laissant aux développeurs le choix de l'algorithme permettant de les résoudre. Ensuite, ces bibliothèques peuvent être utilisées pour le développement d'environnements de plus haut niveau (par exemple, d'autres bibliothèques ou des logiciels mathématiques).

1.2 La bibliothèque BLAS

BLAS (Basic Linear Algebra Subroutines) ([72], [74]) est une collection de routines C et FORTRAN pour les opérations de base en algèbre linéaire numérique. Elle fournit un ensemble de routines très robustes et portables permettant de développer des algorithmes très performants. Plus précisément, il existe trois niveaux d'opérations BLAS :

1) **Le niveau 1 (BLAS 1)** Ce niveau regroupe les opérations sur les vecteurs (vecteurs- vecteurs). Par exemple, le produit scalaire de deux vecteurs. Ces opérations contiennent un niveau de boucle.
2) **Le niveau 2 (BLAS 2)** Ce niveau regroupe les opérations entre matrices et vecteurs. Par exemple, le produit matrice – vecteur. Ces opérations contiennent deux niveaux de boucles.
3) **Le niveau 3 (BLAS 3)** Ce niveau concerne les opérations entre matrices. Par exemple, le produit de deux matrices. Ces opérations contiennent trois niveaux de boucles.

La bibliothèque BLAS est un standard incontournable dans le calcul scientifique, elle est utilisée dans la plupart des grands codes numériques. Elle est également utilisé comme base dans le développement d'autres bibliothèques comme LAPACK (Linear Algebra PACKage) [75].

1.2.1 Le produit matriciel [67] [68] [71]

Soient deux matrices A et B telles que A (m, n) et B (n, K). Le produit de ces deux matrices est une matrice C (m, K), dont les facteurs C_{ik} sont calculés par la formule suivante:

$$C_{ik} = \sum_{j=0}^{n} A_{ij} * B_{jk} \quad \text{Avec i = 1,…, m et k = 1,…, K} \qquad (1)$$

L'un des principes constitutifs des BLAS 3 est d'effectuer les produits matriciels par blocs de petites dimensions, de sorte que les arguments pour un produit de bloc tiennent dans la mémoire cache, limitant ainsi les surcoûts d'accès mémoire. Voici une illustration du produit par blocs :

Considérons les matrices A, B et C, respectivement de taille n1*n2, n2*n3 et n1 *n3 et qu'elles ont repartitionnées en blocs de tailles respectives m1*m2, m2*m3 et m1*m3. On suppose $n_i = m_i * k_i$ pour tout i = 1, 2, 3. L'opération $C = A * B$ peut s'écrire par blocs :

Pour i = 1 à i = k1
Pour k = 1à k = k2
Pour j = 1à j = k3
*Cij = Cij + Aik * Bkj;*
Fin;

Avec, pour la matrice A (resp. B et C) l'écriture Aij (resp. Bij et Cij) représente le bloc Aij (resp. Bij et Cij).

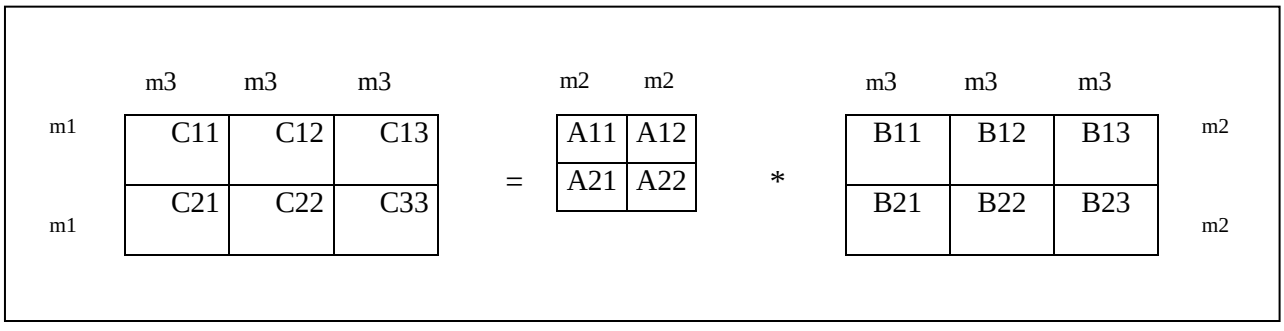

Figure 5.1 : Multiplication matricielle par blocs

Dans la nomenclature des BLAS, l'opération de multiplication matricielle s'appelle GEMM, pour GEneral Matrix Multiplication. Le nom de la routine BLAS s'obtient en y apposant le préfixe correspondant au type utilisé pour la représentation flottante : DGEMM pour des flottants double précision, SGEMM pour des flottants simple précision.

L'opération DGEMM effectue le calcul $C \leftarrow \alpha\ op\ (A) * op\ (B) + \beta\ C$, avec α et β des scalaires, en fonction des arguments qui lui sont transmis. Sa syntaxe est la suivante :

DGEMM (transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc);

Où *op (A*) est une matrice *(m*k)*, *op (B)* est une matrice *(k*n)*, et *C* une matrice *(m*n)* avec : *op (A) = A* si *transa = 'n'* et *op (A) = A^t* si *transa = 't'*. La même explication est valable pour *op (B).*

L'entier *lda* est la dimension principale *(leading dimension)* de la matrice *A*. il détermine comment est stockée la matrice *A* dans le tableau à une dimension *a (par lignes ou par colonnes)*: un élément A_{ij} est rangé à la position *i+lda*j* de *a*. De façon similaire, *ldb* et *ldc* détermine la manière dont sont rangées respectivement les matrices *B* et *C* dans les tableaux *b* et *c*.

Le produit matriciel est une opération qui est utilisée dans de nombreux noyaux de calculs numériques. C'est l'opération la plus utilisée dans les BLAS 3. En effet, l'efficacité des BLAS est principalement basée sur un noyau de calcul optimisé pour le produit matriciel.

1.2.2 La résolution des systèmes [67] [68] [73]

La résolution d'un système AX = B, pour des matrices denses, est essentiellement basée sur la méthode de pivot de GAUSS. Celle-ci peut être décomposé en deux phases distinctes : la mise sous forme triangulaire de la matrice A et la résolution du système triangulaire obtenu.

La résolution d'un système triangulaire se décline sous plusieurs variantes : résolution à gauche, résolution à droite d'une matrice triangulaire supérieure ou inférieure. Nous nous intéressons à la résolution de systèmes triangulaires gauche avec une matrice triangulaire supérieure. Nous avons donc, pour une matrice *A (n, n)*, une matrice *B (n, k)* et une inconnue *X (n, k)* le système *AX = B* suivant :

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \ldots a_{1n} \\ & a_{22} & a_{23} & a_{2n} \\ & & \ldots & \ldots. \\ & & & a_{nn} \end{bmatrix} \times \begin{bmatrix} x_{11} & x_{12} & \ldots & x_{1k} \\ x_{21} & x_{22} & \ldots & x_{2k} \\ \ldots & & \ldots & \ldots \\ x_{n1} & \ldots & \ldots & x_{nk} \end{bmatrix} = \begin{bmatrix} b_{11} & b_{12} & \ldots & b_{1k} \\ b_{21} & b_{22} & & b_{2k} \\ \ldots & \ldots & & \ldots \\ b_{n1} & \ldots & \ldots & b_{nk} \end{bmatrix}$$

La résolution de ce système est donnée par la formule suivante :

$$x_{nl} = \frac{b_{nl}}{a_{nn}} \ , \ x_{il} = \frac{1}{a_{ii}}(b_{il} - \sum_{j=i+1}^{n} a_{ij} x_{jl}) \quad \textit{Avec l=1…k, i=1…n} \quad (2)$$

De plus, si on se ramène au cas d'un système avec matrice à diagonale unitaire, la résolution sera donnée par la formule suivante :

$$x_{nl} = b_{nl} \ , \ x_{il} = (b_{il} - \sum_{j=i+1}^{n} a_{ij} x_{jl}) \quad \textit{Avec l=1…k, i=1…n} \quad (3)$$

Dans la nomenclature BLAS, la résolution des systèmes triangulaire s'appelle TRSM pour TRiangular System solving with Matrix right/left handside. De la même façon que pour le produit matriciel, le préfixe *d* indique que la routine DTRSM correspond à l'implémentation spécialisée pour les flottants double précision. Sa syntaxe est :

DTRSM (side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb);

Cette routine effectue l'un des calculs suivants selon les arguments qui lui sont transmis : $B \leftarrow \alpha.\ op\ (A^{-1}).\ B$ ou $B \leftarrow \alpha.\ B.\ op\ (A^{-1})$.Les arguments *a* et *b* indiquent les vecteurs où sont rangées respectivement les matrices A et B. Les entiers *lda* et *ldb* indiquent la manière du rangement des deux matrices. Et aussi, nous avons :

- L'argument « side » indique si nous avons une résolution gauche ou droite ;
- L'argument « uplo » indique si la matrice est triangulaire supérieure ou inférieure ;
- L'argument « diag » indique si la diagonale de A est unitaire ou non ;

Une approche « diviser pour régner » permet de construire un algorithme par blocs pour ce problème, et de réduire le problème à des produits matriciels profitant ainsi de la performance du produit matriciel par blocs pour la résolution des systèmes triangulaires.

Le produit matriciel et la résolution des systèmes triangulaires des BLAS sont des opérations très utilisées dans le calcul scientifique et constituent un sujet de recherche très vaste ([67], [68], [71], [72]). Il serait donc intéressant d'utiliser les méthodes formelles pour certifier les programmes implémentant ces opérations.

2. Preuve de correction de DGEMM_PLUS et DTRSM

Considérons le programme en C suivant :

```
const int N = 15, K = 1, LDA = 15, LDX = 15;

void DGEMM_PLUS (double A [N * N * LDA], double X[K * LDX], double Y[K * LDX]) {
  for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
      for (int l = 0; l < K; l++)
                Y [i*LDX+l] = Y [i*LDX +l] + A [i*LDA +j] * X [j*LDX +l];
}
void DTRSM (double A [N * N * LDA], double X [K * LDX]) {
  for (int i = N - 2; i >= 0; i--)
    for (int j = i+1; j < N; j++)
      for (int l = 0; l < K; l++)
                X [i*LDX +l] = X [i*LDX +l] – A [i*LDA +j] * X [j*LDX +l];}
```

Figure 5.2 : Le programme C à prouver

Ce programme définit deux fonctions : DGEMM_PLUS et DTRSM.

- ***DGEMM_PLUS*** calcule le produit de deux matrices A et X. Le résultat de ce produit est rangé dans Y. Les matrices dans ce programme sont représentées par des tableaux. DGEMM_PLUS effectue le calcul représenté par la formule (1), page (67)
- ***DTRSM*** résout le système: AX=B. La matrice A est triangulaire supérieure et sa diagonale est unitaire. Le résultat de ce programme est rangé dans le deuxième membre de l'égalité (B). C'est pour cette raison que cette fonction prend en argument seulement deux vecteurs A et X au lieu de trois. DTRSM effectue le calcul représenté par la formule (3), page (68).

2.1 Description du problème

Dans ce travail, nous nous intéressons à démontrer, en utilisant la méthode Why que :

- le programme DGEMM_PLUS calcule bien le produit de matrices ;
- DTRSM calculent bien, la solution du système AX=B, avec A une matrice triangulaire supérieure à diagonale unitaire. Ceci se traduit par la démonstration de la propriété suivante :

$$\text{DGEMM\_PLUS (A, DTRSM (A, X), Y)} = X$$

Les éléments de Y initialement nuls. Ce qui signifie : L'application de DGEMM_PLUS au résultat de DTRSM sur, les vecteurs A et X, retourne le vecteur X initialement soumis à DTRSM.

Pour arriver à ce résultat, les étapes suivantes sont nécessaires :

1- Insérer les annotations concernant la gestion des tableaux et déterminer les variants et invariants des boucles ainsi que les postconditions dans les deux programmes DGEMM_PLUS et DTRSM. De ces annotations découlent les preuves de validations des accès aux tableaux, de la terminaison et de la correction des calculs.

2- Démontrer les obligations de preuves concernant la gestion des tableaux, les variants et invariants des boucles Pour DGEMM_PLUS et DTRSM.

3- Démontrer la postcondition de DGEMM_PLUS.

4- Démontrer la postcondition de DTRSM.

2.2 Annotation et extraction des obligations de preuve des programmes DGEMM_PLUS et DTRSM

2.2.1 Annotation du programme DGEMM_PLUS

Le programme DGEMM_PLUS annoté, en utilisant Caduceus, est donné par la figure (5.4).

1) Pour spécifier que les accès aux tableaux sont valides, nous utilisons le prédicat *valid_range* de Caduceus. L'écriture *\valid_range (t, 0, n-1)* veut dire que tout les emplacements mémoire t[i] pour 0<= i <= n-1 sont valides.

2) Pour l'annotation complète du programme, il est nécessaire de préciser les variants et invariants de chaque boucle ainsi que la postcondition du programme.

a) Variant et invariant de la boucle l

Le variant de cette boucle est *k-l* et son invariant exprime que: *(0<=l<=k)* et nous utilisons le prédicat *is_loop_k*, préalablement déclaré dans les annotations, ici pour exprimer que ou bien la boucle n'a pas encore commencé *(l==0)*, ou bien la boucle a commencé et le prédicat exprime qu'à une étape *l*, le programme a effectué le calcul de la boucle pour toutes les valeurs l_1 telles que $l_1<l$.

b) Variant et invariant de la boucle j

Le variant de cette boucle est *n-j*, et son invariant exprime que :

- *(0<=j<=n) ;*
- à l'itération *j*, le programme a modifié l'élément initial *y [i*ldx+0]* en ajoutant à la valeur initiale exprimé par *(\old y [i*ldx+0])* la quantité :

$v = \sum_{j1=0}^{(j-1)} A[i*lda+j1]*X[j1*ldx+0]$, calculée avec la fonction *sum_prod_j_n* préalablement déclarée dans les annotations.

c) Variant et invariant de la boucle i

Le variant de cette boucle est *n*-i et son invariant exprime que :

- *(0<=i<=n) ;*
- à une itération *i*, le programme a modifié l'élément *Y[i_1*ldx +0]* pour *0<=i_1<i*, en ajoutant à la valeur initiale la quantité :

$$v = \sum_{j=0}^{n-1} A[i_1 * lda + j] * X[j * ldx + 0]$$

Finalement, la clause loop_assigns spécifie que seuls les éléments du vecteur Y sont modifiés dans ce programme et particulièrement par les boucles sur k et sur j.

3) La postcondition du programme DGEMM_PLUS est la relation :

$$Y[i * ldx + 0] := @Y[i * ldx + 0] + \sum_{j=0}^{n-1} A[i * lda + j] * X[j * ldx + 0]$$

Pour chaque i, $0 \leq i < n$. Avec, l'écriture *@Y[i*ldx+0]* qui signifie: la valeur *Y[i*ldx+0]* avant l'exécution de la fonction.

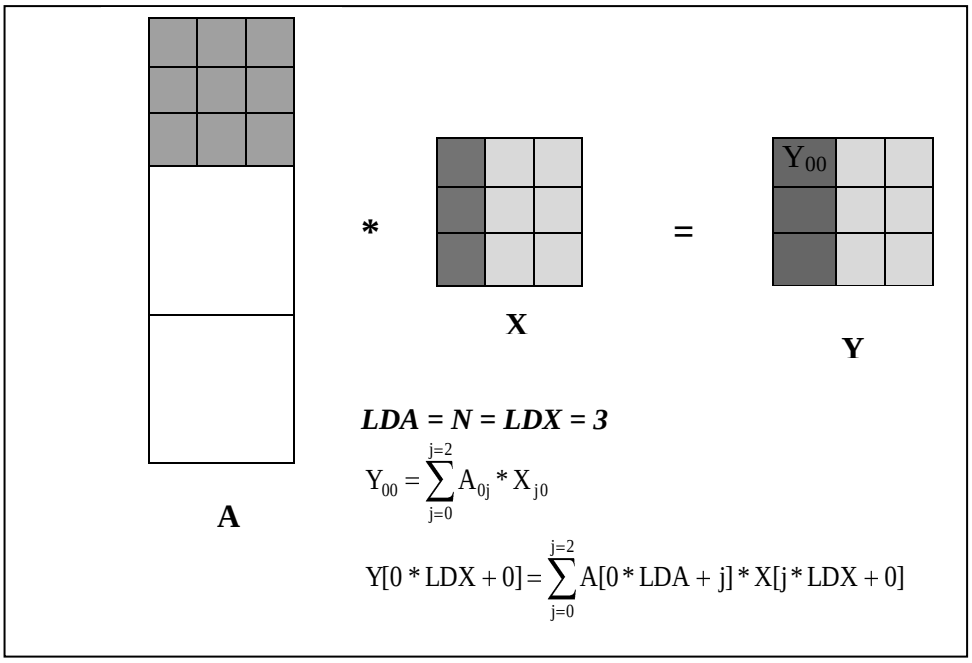

Figure 5.3 : Illustration du calcul DGEMM_PLUS

```
const int N = 15, K = 1, LDA = 15, LDX = 15;
/*@ predicate is_loop_k (double* a, double* x, double* y, int n,int m,  int k,  int lda, int ldx )
@  reads x[..] , a[..], y[..]
@*/
/*@ logic double sum_prod_j_n (double* a, double* x, int i,int j,  int k,int m,  int lda, int ldx )
@  reads x[..] , a[..]
@*/
/*@ requires \valid_range (A, 0, (((n-1)*lda) +( n-1)))     &&
@ \valid_range (X,0 , (((n-1)*ldx) +( k-1)))     &&
@ \valid_range (Y, 0, (((n-1)*ldx)+( k-1)))    &&
@ (0<n) && (0<k) && (lda>0) && (ldx>0)    &&
@ (A!=Y) && (X!=Y) && (A!=X)
@
@  ensures
@  (\forall int i;  (0<=i<n) => Y[(i)*ldx+0] == (\old(Y[(i)*ldx+0])) +
@   sum_prod_j_n (A, X,  (i),0, 0, (n-1) , lda, ldx))
@*/

void DGEMM_PLUS (double A [N * N * LDA], double X[K * LDX], double Y[K * LDX]) {
/*@ invariant i>=0 && i<=n && (( \forall int i1; (0<=i1<i) =>  Y[(i1)*ldx+0]==
@((\old(Y[(i1)*ldx+0]))+
@sum_prod_j_n  (A, X,  (i1) ,0, 0, (n-1) , lda, ldx))))
@ variant n-i
@*/
  for (int i = 0; i < N; i++)
{
/*@ assert (\forall int i1;(i1==i)=> Y[i*ldx+0]== (\old(Y[i1*ldx+0])))
@*/
/*@ invariant j>=0 && j<=n && (\forall int i1; (i1==i)=> Y[i*ldx+0]==
@((\old(Y[i1*ldx+0]))+
@sum_prod_j_n  (A, X,  I, 0, 0,(j-1), lda, ldx)))
@ loop_assigns *Y
@ variant n-j@*/
   for (int j = 0; j < N; j++){
/*@invariant l>=0 && l<=k  && (l==0  || is_loop_k (A, X, Y, i, j, (l-1), lda, ldx))
@ loop_assigns *Y
@ variant k-l@*/
     for (int l= 0; l < K; l++){
            Y [i*LDX+l] = Y [i*LDX +l] + A [i*LDA +j] * X [j*LDX +l];} } } }
```

Figure5.4: Annotation du programme DGEMM_PLUS

2.2.2 Annotation du programme DTRSM

Le programme DTRSM annoté est donné par la figure (5.5).

1) Pour spécifier que les accès aux tableaux sont valides, nous utilisons le prédicat *valid_range.* On doit exprimer également dans la précondition que le tableau A représente une matrice triangulaire supérieure à diagonale unitaire.

Ces deux propriétés, sont définies respectivement à l'aide des deux prédicats *triang_sup* et *diag_un.*

2) Les variants et invariants de boucles pour DTRSM sont déterminés comme ceci :

a) Variant et invariant de la boucle l

Le variant de cette boucle est *k-l* et son invariant exprime que nous avons *(0<=l<=k)* et nous utilisons le prédicat *is_loop_k_2* ici pour exprimer que ou bien la boucle n'a pas encore commencé *(l==0),* ou bien la boucle a commencé et le prédicat exprime qu'à une étape *l,* le programme a effectué le calcul de la boucle pour toutes les valeurs $l_1 < l$.

b) Variant et invariant de la boucle j

Le variant de cette boucle est *n-j* et son invariant exprime que :

- *(i+1) <=j<=n ;*
- à l'itération *j,* le programme a modifié l'élément initial *X[i*ldx+0]* en faisant soustraire à la valeur initiale la valeur v :

$v = \sum_{j1=i+1}^{(j-1)} A[i * lda + j1] * X[j1 * ldx + 0]$, calculée avec la fonction *sum_prod_j_n.*

c) Variant et invariant de la boucle i

Le variant de cette boucle est *i* et son invariant exprime que nous avons *(-1)<=i<=(n-2)* et qu'à une itération *i,* le programme a modifié l'élément $X[i_1 * ldx + 0]$ pour $i < i_1 <= (n-2)$,en faisant soustraire à la valeur initiale la valeur v :

$v = \sum_{j=i1+1}^{(n-1)} A[i1 * lda + j] * X[j * ldx + 0]$

3) Concernant la postcondition, nous avons la condition nécessaire et suffisante pour que le programme DTRSM calcule bien la solution du système AX=B est la relation :

DGEMM_PLUS (A, DTRSM (A, X), 0) = X.

La démonstration de cette propriété est présentée au paragraphe (2.3).

```
const int N = 15, K = 1, LDA = 15, LDX = 15;
/*@ predicate is_loop_k_2 (double* a, double* x, double* y, int n,int m,  int k,  int lda, int ldx )
@  reads x[..] , a[..]
@*/
/*@ logic double sum_prod_j_n (double* a, double* x, int i,int j,  int k,int m,  int lda, int ldx )
@  reads x[..] , a[..]
@*/
/*@ predicate triang_sup  (double* a,int I, int j )  reads  a[..] @*/
/*@ predicate diag_un (double* a, int i, int j )  reads  a[..] @*/

/*@ requires \valid_range (A, 0,( ((n-1)*lda) +( n-1)))    &&
@ \valid_range (X,0 ,( ((n-1)*ldx) +( k-1) ))                &&
@  (2<=n) && (0<k) && (ldx>0) && (lda>0)   &&
@ triang_sup (A, n, lda) && diag_un (A, n, lda)
@ (A!=X)
@*/

void DTRSM (double A [N * N * LDA], double X [K * LDX]) {

/*@ invariant i>=(-1) && i <= (n-2) &&   (( \forall int i1; (i<i1<=(n-2)) =>  X[(i1)*ldx+0]==
((\old(X[(i1)*ldx+0]))-sum_prod_j_n  (A, X,  (i1) ,(i1+1), 0, (n-1), lda, ldx)))) &&
@  triang_sup (A, n, lda) && diag_un (A, n, lda)
@ variant i
@*/
  for (int i = N - 2; i >= 0; i--){
/*@ assert (\forall int i1;(i1==i)=> X[i*ldx+0]== (\old(X[i1*ldx+0])))
@*/
/*@ invariant j>=i+1 && j<=n && (\forall int i1; (i1==i) => X[i*ldx+0]== \old (X[i1*ldx+0])
@ - sum_prod_j_n (A, X ,  i, (i+1),  0 , (j-1) ,   lda,  ldx ))&& (@ triang_sup (A, n, lda) &&
@diag_un (A, n, lda)
@ loop_assigns X[i*ldx+0]
@ variant n-j
@*/
    for (int j = i+1; j < N; j++){
/*@ invariant l>=0 && l<=k&&(l==0  || is_loop_k_2 (A, X, X, i, j, (l-1), lda, ldx))
@ loop_assigns X[i*ldx+0] &&  triang_sup (A, n, lda) && diag_un (A, n, lda)
@ variant k-l@*/
  for (int l= 0; l < K; l++)
{X [i*LDX +l] = X [i*LDX +l] – A [i*LDA +j] * X [j*LDX +l];}}}}
```

Figure 5.5 : Annotation du programme DTRSM

2.2.3 Application de Caduceus et Why

L'application de l'outil Caduceus puis Why sur ces fonctions produit des obligations de preuves que l'on peut classer en quatre catégories:

1- des obligations de preuves concernant la validité de chaque invariant à l'entrée et après chaque itération de sa boucle ;
2- la validité des accès aux tableaux;
3- la terminaison de chaque boucle;
4- la validité de la postcondition de DGEMM_PLUS.

L'option –no—fp

Dans notre travail, nous exécutons l'outil Caduceus pour les programme DGEMM_PLUS et DTRSM avec l'option *–no—fp*. Ceci veut que nous ne prenons pas en compte le modèle des nombres flottant de Why et que le type double des programmes C sera confondu avec les nombres réels du système Coq.

2.2.4 Définition des objets logiques dans Coq

Les figures 5.6, 5.7 et 5.8 montrent respectivement les définitions du prédicat *is_loop_k*, de la fonction *sum_prod_j_n* et du prédicat *is_loop_k_2 :*

1 Le prédicat *is_loop_k*

Le prédicat *is_loop_k* a été introduit pour exprimer la propriété de l'invariant sur *l* dans DGEMM_PLUS. Il est défini avec le prédicat inductif *is_loop_k_aux*. Ce prédicat est défini avec deux constructeurs *elt_k_0* et *elt_k_s*.

Le constructeur elt_k_0 exprime que :

Si le calcul Y[i*LDX+k] = Y [i*LDX+k]+A [i*LDA+j]*X [j*LDX+k] est effectué sur l'élément *Y[i*ldx+0*], alors la propriété est vrai pour *k=0*.

Le constructeur *elt_k_s* exprime la façon dont doit se dérouler le même calcul sur les autres éléments pour vérifier la propriété: si le calcul a été effectué jusqu'à *k*, et si on l'effectue à partir de là sur l'élément *Y [i*ldx+(k+1)]* , alors la propriété est vérifiée pour *k+1*.

2 La fonction *sum_prod_j_n*

La fonction *sum_prod_j_n* calcule à l'aide de la fonction récursive *som_prod_aux* la valeur $v = \sum_{j0=i1}^{j} A[i*lda+j0]*X[j0*ldx+0]$

3 Le prédicat *is_loop_k_2*

Le prédicat *is_loop_k_2* a été introduit pour exprimer la propriété de l'invariant sur *l* dans DTRSM. Il est défini avec le prédicat inductif *is_loop_k_aux_2*. Ce prédicat est défini avec deux constructeurs *elt_k_0_2* et *elt_k_s_2*.

Le constructeur elt_k_0_2 exprime que :

Si le calcul X [i*LDX +k] = X [i*LDX +k] – A [i*LDA +j] * X [j*LDX +k] est effectué sur l'élément *X[i*ldx+0*], alors la propriété est vrai pour *k=0*.

Le constructeur *elt_k_s* exprime la façon dont doit se dérouler le même calcul sur les autres éléments pour vérifier la propriété: si le calcul a été effectué jusqu'à *k*, et si on l'effectue à partir de là sur l'élément *X [i*ldx+(k+1)]* , alors la propriété est vérifiée pour *k+1*.

```
Inductive is_loop_k_aux ( a x y: pointer global) ( i j lda ldx:Z): memory R global->Z->Prop:=

| elt_k_0: forall k:Z, forall mem, (exists mem0, (shift y (i*ldx+ 0) #mem ) =((shift y (i*ldx+ 0) #mem0) +
((shift a (i * lda+j) #mem0) *(shift x (j * ldx+0)#mem0)))%R)->k=0-> is_loop_k_aux a x y i j lda ldx mem k

| elt_k_s: forall k:Z, forall mem:memory R global, is_loop_k_aux a x y i j lda ldx  mem k ->
is_loop_k_aux a x y i j lda ldx ( upd mem (shift y (i*ldx+(k+1) ) ) (( shift y (i*ldx+(k+1) )#mem) +
((shift a (i * lda+j) #mem) *(shift x (j * ldx+ (k+1))#mem)))%R)  (k+1).

 Definition is_loop_k :   (memory R global) -> (pointer global) -> (pointer global)  -> (pointer global) -> Z ->
Z -> Z -> Z -> Z -> Prop.
  intros mem a x y i j k lda ldx.
  exact (is_loop_k_aux  a x y i j lda ldx mem k). Defined.
```

Figure 5.6 : Définition du prédicat is_loop_k

```
Fixpoint som_prod_aux (a x:pointer global) (mem:memory R global) (i j l lda ldx:Z) (n:nat) {struct n}:R:=
   match n with
   | 0%nat => (shift a (i*lda+ j) #mem * (shift x (j*ldx+ l) #mem ))%R
   | S n1 =>Rplus (( ( shift a (i*lda+j) #mem )* (shift x (( j)*ldx+ l) #mem ))%R)
                    (som_prod_aux  a x mem i (j-1) l  lda ldx n1)%R

end.

Definition sum_prod_j_n (mem:memory R global) (a x:pointer global) (i i1  l j lda ldx:Z):= match ( j ?= (i1))
with

| Lt => 0%R
| _ => (som_prod_aux a x mem i j l   lda ldx  (Zabs_nat (j -( i1))%Z))%R
end.
```

Figure5.7 : Définition de la fonction sum_prod_j_n

```
Inductive is_loop_k_aux_2 ( a x y: pointer global) ( i j lda ldx:Z): memory R global->Z->Prop:=
| elt_k_0_2: forall k:Z, forall mem, (exists mem0, (shift y (i*ldx+ 0) #mem ) =((shift y (i*ldx+ 0) #mem0) -
((shift a (i * lda+j) #mem0) *(shift x (j * ldx+0)#mem0)))%R)->k=0-> is_loop_k_aux_2 a x y i j lda ldx mem
k
| elt_k_s_2: forall k:Z, forall mem:memory R global, is_loop_k_aux_2 a x y i j lda ldx  mem k ->
is_loop_k_aux_2 a x y i j lda ldx ( upd mem (shift y (i*ldx+(k+1) ) ) (( shift y (i*ldx+(k+1) )#mem) - ((shift a
(i * lda+j) #mem) *(shift x (j * ldx+ (k+1))#mem)))%R)  (k+1).

Definition is_loop_k_2 :
 (memory R global) -> (pointer global) -> (pointer global)
 -> (pointer global) -> Z -> Z -> Z -> Z -> Z -> Prop.
 intros mem a x y i j k lda ldx.
 exact (is_loop_k_aux_2  a x y i j lda ldx mem k).
 Defined.
```

Figure5.8: Définition du prédicat is_loop_k_2

2.2.5 Les obligations de preuves

Nous avons déchargé les obligations de preuves dans le système Coq comme suit :

Les obligations de preuves concernant la terminaison (au nombre de six) des boucles ont été effectuées, pour les deux programmes DGEMM_PLUS et DTRSM, par l'outil lui-même (six preuves de terminaison).

Les obligations de preuves concernant la gestion de la validité des accès aux tableaux (au nombre de huit, quatre accès pour chaque fonction) de sont déchargées par simple substitution de variables en utilisant les tactiques suivantes : *(intuition ; subst ; auto.).*

Les preuves des invariants représentent la plus grande partie de travail de démonstration. Après avoir défini les objets logiques, nous avons démontré la validité de chaque invariant (au nombre de six dans les deux programmes) à l'entrée de sa boucle ainsi qu'une démonstration de sa préservation après chaque itération, ce qui fait douze obligations de preuves.

La démonstration de la postcondition de DGEMM_PLUS découle directement de la démonstration de l'invariant de la boucle sur *i* de cette fonction : dans cet invariant nous avons :

*((\forall int i1; (0<=i1<i) => Y[(i1)*ldx+0]== @((\old(Y[(i1)*ldx+0]))+*
@sum_prod_j_n (A, X, (i1) ,0, 0, (n-1) , lda, ldx))))

A la fin de cette boucle, on sort avec la variable *i* qui vaut *N.* En remplaçant i par N, on obtient :

*((\forall int i1; (0<=i1<N) => Y[(i1)*ldx+0]== @((\old(Y[(i1)*ldx+0]))+*
@sum_prod_j_n (A, X, (i1) ,0, 0, (n-1) , lda, ldx))))

Ce qui correspond exactement à la postcondition de DGEMM_PLUS.

2.3 Preuve dans Coq de la relation DGEMM_PLUS (A, DTRSM (A, X), 0) = X.

Pour démontrer dans Coq la relation DGEMM_PLUS (A, DTRSM (A, X), 0) = X, il est nécessaire de :

1- Définir dans coq un type matrice qui prenne en considération le modèle mémoire Caduceus;
2- définir l'objet logique dgemm_plus: une fonction sur ce type matrice ;
3- démontrer que la fonction dgemm_plus de Coq réalise bien la fonction C DGEMM_PLUS;
4- écrire la propriété DGEMM_PLUS (A, DTRSM (A, X), 0)=X et la démontrer une fois les obligations de preuves générées.

2.3.1 Présentation du type matrice

Nous proposons de définir le type matrice *(Matrix)* de façon axiomatique. La figure 5.7 montre la définition du type matrice et les différents axiomes et fonctions associés à ce type.

2.3.1.1 Définition

***Definition matrix**: Z->Z->Set->Set*

Cette ligne permet de déclarer le type matrice. Les paramètres sont respectivement : le nombre de lignes, le nombre de colonnes et le type des éléments de la matrice. Le type *matrix* est logé dans le type des objets calculatoire Set. Selon cette définition, l'écriture: *(A: matrix n m R)* veut dire que *A* est de type matrice, le nombre de ses lignes est *n*, le nombre de ses colonnes et *m* et le type des éléments qu'elle contient est *R*.

2.3.1.2 Fonctions d'accès et de mise à jour

La fonction permettant d'accéder à un élément d'une matrice est déclarée comme suit :

***Definition acc_mat**: forall (A:Set),forall (n m :Z), (matrix n m A)->Z->Z-> A.*

On accède à un élément d'une matrice *(matrix n m A)* en donnant le numéro de sa ligne et celui de sa colonne. Cette fonction retourne un résultat de type *A.*

La fonction permettant la mise à jour d'une matrice est déclarée comme suit :

***Definition updte_mat**: forall (A:Set), forall (n m :Z),(matrix n m A) ->A->Z->Z-> (matrix n m A).*

Cette fonction permet de modifier un élément d'une matrice *(matrix n m A)* en donnant le numéro de sa ligne, le numéro de sa colonne et la nouvelle valeur de l'élément concerné. Cette fonction retourne une nouvelle matrice *(matrix n m A).*

L'axiome *acc_updte_mat* est relatif aux fonctions d'accès et de mise à jour. Il stipule que l'accès dans une matrice *mat* telle que *(mat :matrix n m A)*à un élément qui a les indices *i et j* et dont la valeur a été mise à *v* par la fonction *updte_mat* donne la valeur v.

Axiom acc_updte_mat: *forall (A:Set) ,forall (n m:Z), forall (v:A), forall (mat :matrix n m A), forall (i j :Z), i<= n -> j<= m ->(acc_mat (updte_mat mat v i j) i j)=v.*

2.3.1.3 Passage vers le type *pointer*

La fonction *mat_to_ptr* permet d'obtenir à un état mémoire donnée, une représentation pointer pour une matrice donnée. Cette fonction nous permettera de raisonner sur le nouveau type *Matrix* tout en nous permettant de garder le lien avec le modèle mémoire de Caduceus qui est basé sur le type pointer.

Definition mat_to_ptr*: forall (A:Set),forall (n m :Z), (matrix n m A)->(memory A global) -> (pointer global).*

L'axiome *length_ptr_mat* concerne la taille du block *pointer* qui est associé à une matrice (mat :matrix n m A).

Axiom length_ptr_mat*: forall (A:Set), forall (a:alloc_table) ,forall (mem: memory A global), forall (p:pointer global), forall (n m :Z), forall (mat: matrix n m A), (n>0) ->(m>0)-> (p= (mat_to_ptr mat mem))-> (block_length a p)= (n*m).*

L'axiome *elts_ptr_mat* concerne la correspondance entre les éléments d'une matrice et les éléments du pointer qui la représente à un état mémoire donné. On suppose que la matrice est parcourue ligne par ligne.

Axiom elts_ptr_mat: forall (A:Set), forall (mem: memory A global) ,forall (n m:Z), forall (mat:matrix n m A), forall (p: pointer global), (p= mat_to_ptr mat mem)->forall (i j:Z),(n>0) ->(m>0)->
(0<=(i) < (n)) -> (0<= (j)< (m)) -> acc_mat mat i j = shift p ((i)*(m)+(j)) # mem.

```
Require Export Caduceus.
Definition global:Set.
Admitted.

Definition matrix: Z->Z->Set->Set.
Definition acc_mat: forall (A:Set),forall ( n m :Z), (matrix n m A)->Z->Z-> A.
Definition updte_mat: forall (A:Set) , forall ( n m :Z),( matrix n m A) ->A->Z->Z-> (matrix n m A).

 Axiom acc_updte_mat : forall (A:Set) ,forall (n m:Z), forall (v:A), forall (mat :matrix n m A),
forall (i j :Z),(n>0)-> (m>0)-> (i<=n) -> (j <= m) ->(acc_mat (updte_mat mat v i j) i j)=v.

Definition mat_to_ptr: forall (A:Set),forall (n m :Z) , (matrix n m A)->(memory A global) -> (pointer
global).

Axiom length_ptr_mat: forall (A:Set), forall (a:alloc_table) ,forall (mem: memory A global),
      forall (p:pointer global), forall (n m :Z), forall (mat: matrix n m A), (n>0) ->(m>0)->
            (p= (mat_to_ptr mat mem))-> (block_length a p)= (n*m).

Axiom elts_ptr_mat: forall (A:Set), forall (mem: memory A global) ,forall ( n m:Z ),
 forall (mat:matrix n m A), forall ( p: pointer global), (p= mat_to_ptr mat mem )->forall (i j:Z ), (n>0)->
(m>0)-> (0<=( i) < ( n)) -> (0<= ( j)< ( m)) -> acc_mat mat i j = shift p ((i)*( m)+(j)) #mem.
```

Figure 5.9 : Définition du type matrice dans Coq

2.3.2 Définition de la fonction dgemm_plus en Coq

Pour définir la fonction *dgemm_plus*, on utilise une fonction *som_prod_mat* qui calcule la somme *v* suivante sur le type matrice :

$$v = \sum_{j=0}^{n-1} A[i * lda + j] * X[j * ldx + 0]$$

C'est l'équivalent de la fonction *sum_prod_j_n* sur le type *pointer.* Elle est donnée par une définition récursive (figure 5.10)

```
Fixpoint  som_prod_mat (n k lda ldx :Z) (a  : matrix (n*n)  lda  R)
 (x: matrix  ldx k R) (i  l:Z) (m:nat) {struct m} :R :=
match m with
| 0%nat =>  ((((acc_mat a i  (Z_of_nat m) ))* (acc_mat  x   (Z_of_nat m) l))%R)
| S m1 =>  (((((acc_mat  a i   (Z_of_nat m)))* (acc_mat  x  (Z_of_nat m)  l))%R)
            + (som_prod_mat   n  k  lda ldx a x i l m1  )%R)%R
end.
```

Figure 5.10 : La fonction som_prod_mat

La fonction *dgemm_plus* (figure 5.11) est définie à l'aide de la fonction *dgemm_plus_aux*. Cette dernière est une fonction récursive qui effectue les appels récursifs sur l'argument *n1* pour parcourir tous les éléments de *y* à modifier avec la fonction. Ces appels récursifs représentent les valeurs *i* de la boucle de la fonction C DGEMM_PLUS. Les éléments sont modifiées en les mettant à jour (fonction *updte_mat*) par les valeurs correspondantes calculées avec la fonction *som_prod_mat.*

```
Fixpoint dgemm_plus_aux  (n  k lda ldx :Z) ( a:matrix (n* n) lda  R)  ( x y : matrix  ldx k R)
          ( n1 m :nat) {struct n1} : (matrix  ldx k R):=
match n1 with
| 0%nat => (updte_mat y (som_prod_mat n k lda ldx a x (Z_of_nat n1) 0  m )%R  (Z_of_nat n1) 0
)
| S n2 => let yt := (updte_mat  y  ((som_prod_mat n k lda ldx   a x (Z_of_nat n1) 0  m  ))%R
(Z_of_nat n1) 0 ) in
  dgemm_plus_aux  n k lda ldx  a x yt n2  m
end.

  Definition dgemm_plus (n k lda ldx m:Z ) ( a:matrix (n* n) lda  R)  ( x y : matrix  ldx k R) :
 (matrix  ldx k R).
  intros n k lda ldx m  a x y .
  exact (dgemm_plus_aux  n k lda ldx a x y (Zabs_nat (n-1)) (Zabs_nat m)).
  Defined.
```

Figure 5.11 : La fonction Coq dgemm_plus

2.3.3 Démonstration que dgemm_plus réalise DGEMM_PLUS

Nous allons dans ce qui suit exprimer que la fonction Coq dgemm_plus réalise bien le programme C DGEMM_PLUS avec le prédicat DGEMM_is_dgemm. Nous ajoutons d'abord cette propriété comme postcondition à la fonction DGEMM_PLUS après avoir déclaré le prédicat DGEMM_is_dgemm (figure 5.12).

```
const int N = 15, K = 1, LDA = 15, LDX = 15;
/*@ predicate is_loop_k (double* a, double* x, double* y, int n,int m,  int k,  int lda, int ldx )
@  reads x[..] , a[..], y[..]
@*/
/*@ logic double sum_prod_j_n (double* a, double* x, int i,int j,  int k,int m,  int lda, int ldx )
@  reads x[..] , a[..]
@*/
/*@ predicate DGEMM_is_dgemm (double* a, double* x, double* y, int n,int m,  int k,
@int lda, int ldx ) reads x[..] , a[..], y[..]  @*/

/*@ requires \valid_range (A, 0, (((n-1)*lda) +( n-1)))     &&
@ \valid_range (X,0 , (((n-1)*ldx) + (k-1)))     &&
@ \valid_range (Y, 0, (((n-1)*ldx)+ (k-1)))    &&
@@vect_nul (Y)
@ (0<n) && (0<k) && (lda>0) && (ldx>0)    &&
@ (A!=Y) && (X!=Y) && (A!=X)
                                        @  ensures
@  (\forall int i;  (0<=i<n) => Y[(i)*ldx+0] == (\old(Y[(i)*ldx+0])) +
@   sum_prod_j_n (A, X,  (i),0,0, (n-1), lda, ldx)) &&
@ DGEMM_is_dgemm (A, X, Y, n, n, k, lda, ldx)
@*/
void DGEMM_PLUS (double A [N * N * LDA], double X[K * LDX], double Y[K * LDX]) {…}
```

Figure 5.12 : Annotation de DGEMM_PLUS avec la propriété DGEMM_is_dgemm

Le prédicat DGEMM_is_dgemm est ensuite défini dans Coq comme suit (figure 5.13): si *Yt* est le résultat de l'application de la fonction Coq dgemm_plus sur les matrices *Am* et *Xm*, et si *p* est la représentation pointer de la matrice *Yt*, alors les élément du pointer *p* sont les mêmes que ceux du pointer *y*, résultant de l'application de DGEMM_PLUS en C sur les pointers a et x qui sont les représentations pointer des matrices Am et Xm respectivement.

```
Definition DGEMM_is_dgemm :
  (memory R global) -> (pointer global) -> (pointer global)
  -> (pointer global) -> Z -> Z -> Z -> Z -> Z -> Prop.

intros mem a x y  n m k lda ldx.

  exact ( forall (Am: matrix (n*n) lda R), forall (Xm Ym Yt: matrix  ldx k R ),
 (a = mat_to_ptr  Am mem )->  (x = mat_to_ptr  Xm mem )
  ->(Yt =( dgemm_plus n k lda ldx m Am Xm Ym)) ->
forall( p: pointer global),forall (i:Z), (p= mat_to_ptr Yt mem) -> ( 0<=i<n) ->
  (shift y (i*ldx + 0) #mem) = (shift p (i*ldx +0)#mem)).  Defined.
```

Figure 5.13 : Définition du prédicat DGEMM_is_dgemm

Pour démontrer la propriété DGEMM_is_dgemm, nous démontrons d'abord que la fonction *som_prod_mat* sur les matrices correspond bien à la fonction *sum_prod_j_n* sur les pointers. Ceci est réalisé avec le lemme *som_prod_mat_ptr :*

Le lemme *som_prod_mat_ptr* introduit ci-dessous exprime que si :

- Si A et X sont deux matrices et si p1 et p2 sont leurs « pointers » respectifs ;
- Alors, la valeur obtenue par l'application de *som_prod_mat* sur les éléments des matrices A et X, est celle obtenue par l'application de la fonction *sum_prod_j_n* sur les mêmes éléments de p1 et p2, en prenant bien sur les mêmes arguments pour les bornes des sommes.

Lemma som_prod_mat_ptr*: forall (mem: memory R global), forall (p1 p2: pointer global), forall (n k lda ldx: Z), forall (a: matrix (n*n) lda R), forall (x: matrix ldx k R), (p1 = mat_to_ptr a mem) -> (p2 = mat_to_ptr x mem) -> forall (i :Z),(n>0)-> 0<=i<n ->(som_prod_mat n k lda ldx a x i 0 (Zabs_nat (n-1)))%R = (sum_prod_j_n mem p1 p2 i 0 0 (n-1) lda ldx)%R.*

Nous démontrons ensuite que chaque élément d'une matrice résultant de dgemm_plus, est effectivement calculé avec la fonction *som_prod_mat*. Ceci est réalisé avec le lemme *dgemm_plus_elts.*

Le lemme d*gemm_plus_elts* exprime que :

- Si *yt* est le résultat de l'application de la fonction dgemm_plus sur les matrices a, x et y, alors les éléments de *yt* sont calculés avec la fonction som_prod_mat.

Lemma dgemm_plus_elts*: forall (n k lda ldx :Z), forall (a: matrix (n*n) lda R) ,forall (x y yt : matrix ldx k R),forall i: Z, 0<=i<n-> (yt = dgemm_plus n k lda ldx n a x y) -> ((acc_mat yt i 0) = som_prod_mat n k lda ldx a x i 0 (Zabs_nat (n-1))).*

2.3.4 Démonstration de la propriété DGEMM_PLUS (A, DTRSM (A, X), 0) =X

La propriété DGEMM_PLUS (A, DTRSM (A, X), 0)=X, qui constitue la postcondition de la fonction DTRSM est énoncée avec le prédicat ***property*** déclarée dans les annotations (figure 5.14).

```
/*@ predicate is_loop_k_2 (double* a, double* x, double* y, int n,int m,  int k,  int lda, int ldx )
@  reads x[..] , a[..]
@*/
/*@ predicate diff_ptr (double* a, double* x)
@
@*/
/*@ logic double sum_prod_j_n (double* a, double* x, int i,int j,  int k,int m,  int lda, int ldx )
@  reads x[..] , a[..]
@*/
/*@ predicate triang_sup  (double* a,int I, int j )  reads  a[..] @*/
/*@ predicate diag_un (double* a, int i, int j )  reads  a[..] @*/

/*@ predicate property (double* a, double* x, int n,int k,   int lda, int ldx )  reads x[..] , a[..]
@*/
/*@ requires \valid_range (A, 0,( ((n-1)*lda) + (n-1)))    &&
@ \valid_range (X,0 ,( ((n-1)*ldx) +( k-1) ))                &&
@  (2<=n) && (0<k) && (ldx>0) && (lda>0)   &&
@ triang_sup (A, n, lda) && diag_un (A, n, lda)
@ (A!=X)
@ ensures property (A, X, n ,k, lda,  ldx)
@*/
void DTRSM (double A [N * N * LDA], double X [K * LDX]) {…}
```

Figure 5.14 : Annotation de DTRSM avec sa postcondition

La figure 5.15 représente la définition du prédicat *property* dans Coq. Ce prédicat exprime que :

- si x est le résultat de DTRSM en C sur les tableaux a et x et est la représentation pointer d'une matrice Xm ;
- si a est la représentation pointer d'une matrice Am ;
- Si X1m est le résultat de l'application de dgemm_plus sur Xm et Am ;
- Si x1 est la représentation pointer de X1m ;
- S'il existe un état mémoire mem0 à partir duquel on a effectué le calcul de la foction DTRSM ;
- Alors le pointer x1 correspond au pointer x avant l'execution de DTRSM (à l'état mem0).

```
Definition property :
  (memory R global) -> (pointer global) -> (pointer global) -> Z -> Z -> Z -> Z -> Prop.
intros mem a x  n  k lda ldx.

  exact ( forall (Am: matrix (n*n) lda R), forall (Xm X1m Yt: matrix  ldx k R ), (a = mat_to_ptr  Am
mem )-> (x = mat_to_ptr  Xm mem )
  -> (X1m = (dgemm_plus n k lda ldx n Am Xm Yt))
  -> forall (x1: pointer global), x1 = mat_to_ptr X1m mem
       -> (exists mem0,
             (forall (i1:Z),
               (0 <= i1 /\ i1 <= (n - 2) ->
                (eq (acc mem (shift x (i1 * ldx + 0))) (   Rminus
     (acc mem0 (shift x (i1 * ldx + 0))) (sum_prod_j_n mem a x i1  (i1 + 1) 0 (n - 1) lda ldx))))
       -> (forall (i1:Z), 0<= i1 <= (n-2)
       -> ((acc mem (shift x1  (i1 * ldx + 0))) = (acc mem0 (shift x  (i1 * ldx + 0)))))))).

Defined.
```

Figure 5.15 : Définition du prédicat *property* dans Coq

La démonstration de cette propriété a nécessité les lemmes suivants :

Lemma som_prod_triang_sup_0*: forall (a x:pointer global), forall (i lda ldx n j:Z), forall (mem: memory R global),*
(0<= i< n) -> (j < i) -> (triang_sup mem a n lda) -> (sum_prod_j_n mem a x i 0 0 j lda ldx = 0%R).

Ce lemme exprime que si A est une matrice triangulaire supérieure, on a: Pour tout i et j, si $j < i$ *alors*, $A[i*lda+j]=0$. Ceci conduit à conclure que dans ce cas :

$$\sum_{j=0}^{i-1} A[i*lda+j]*X[j*ldx+0]=0$$

Lemma som_prod_split*: forall (a x:pointer global), forall (mem: memory R global),*
forall (i b c lda ldx:Z), (b < c) -> (b <= i <=c) ->
((sum_prod_j_n mem a x i b 0 c lda ldx)%R = ((sum_prod_j_n mem a x i b 0 (i-1) lda ldx) +(sum_prod_j_n mem a x i i 0 i lda ldx)+(sum_prod_j_n mem a x i (i+1) 0 c lda ldx))%R).

Voici la signification mathématique de ce lemme:

$$\sum_{j=b}^{c} A[i*lda+j]*X[j*ldx+0]=\sum_{j=b}^{i-1} A[i*lda+j]*X[j*ldx+0]+$$

$$\sum_{j=i}^{i} A[i*lda+j]*X[j*ldx+0]+\sum_{j=i+1}^{c} A[i*lda+j]*X[j*ldx+0]$$

Voici la structure générale de la preuve :

1) Posons Y = dgemm_plus (A, DTRSM (A, @X), 0), avec @X, le vecteur X initialement donné comme argument à DTRSM.

Y = dgemm_plus (A, X, 0), X : le vecteur résultant de DTRSM.

2) Pour chaque i, 0 ≤ i< N, d'après la postcondition de DGEMM_PLUS

$$Y[i*ldx+0]=\sum_{j=0}^{n-1} A[i*lda+j]*X[j*ldx+0]$$

3) On applique le lemme *som_prod_split :*

$$Y[i*ldx+0]=\sum_{j=0}^{i-1} A[i*lda+j]*X[j*ldx+0]+A[i*lda+i]*X[i*ldx+0]+$$
$$\sum_{j=i+1}^{n-1} A[i*lda+j]*X[j*ldx+0]$$

4) Nous avons :

$A[i*lda+i]=1$, car la diagonale de A est unitaire, et aussi d'après le lemme *som_prod_triang_sup_0:*

$$\sum_{j=0}^{i-1} A[i*lda+j]*X[j*ldx+0]=0$$

5) On obtient donc:

$$Y[i*ldx+0]=X[i*ldx+0]+\sum_{j=i+1}^{n-1} A[i*lda+j]*X[j*ldx+0]$$

6) On remplace X [i*ldx+0] par sa valeur d'aprés le calcul DTRSM, et spécialement, l'invariant de la boucle i :

$$Y[i*ldx+0]=(@X[i*ldx+0]$$
$$\underbrace{-\sum_{j=i+1}^{n-1} A[i*lda+j]*X[j*ldx+0])+\sum_{j=i+1}^{n-1} A[i*lda+j]*X[j*ldx+0]}_{=0}$$

7) et on obtient : $Y[i*ldx+0]=@X[i*ldx+0]$, ce qui signifie bien que le vecteur Y résultant de l'application de DGEMM_PLUS sur X, tel que X résulte de l'application de DTRSM sur @X, est bien le vecteur initial @X.

Conclusion et perspectives

Conclusion et perspectives

Nous avons effectué dans ce travail la certification de deux fonctions issues du domaine de l'algèbre linéaire. Il s'agit du produit matriciel (DGEMM_PLUS) et de la résolution des systèmes triangulaires (DTRSM). Ces deux opérations sont très utilisées dans de nombreux codes scientifiques. Plus précisément, les deux fonctions certifiées sont tirées de la librairie scientifique BLAS (Basic Linear Algebra Subroutines).

La certification des deux fonctions a été réalisée avec la méthode formelle Why. Cette méthode propose un outils pour annoter les programmes (Caduceus) et un outil, Why, qui permet de générer à partir d'un programme annoté des obligations de preuves qu'il faut décharger dans un système de preuves, nous avons utilisé Coq, pour garantir la correction du programme vis-à-vis les annotations (sa spécification formelle).

L'écriture de la spécification à été réalisé d'abord avec Caduceus, mais comme cet outil fournit une langage basé sur la logique du premier ordre, celui ci ne permet pas d'énoncer des propriété complexes, nous avons spécifié celles -ci dans le système Coq qui offre un langage de spécification très riche basé sur une logique de second ordre.

Nous avons intensivement utilisé le modèle mémoire défini dans l'outil Caduceus. Ce modèle permet de représenter l'état mémoire d'un programme à un instant donné en proposant une modélisation des structures de données et de la mémoire basée sur une idée de Burstall-Bormat. Le modèle mémoire est défini de manière axiomatique.

En plus des preuves formelles, nous avons proposé une axiomatisation du type matrice dans coq. Cette définition prend en compte le modèle mémoire Caduceus. Nous avons utilisé cette axiomatisation dans nos preuves et nous pensons qu'il est possible de l'utiliser pour certifier d'autres programmes.

Ce travail nous a permis de d'envisager quelques futures directions que nous regroupons comme suit:

Les nombres flottants: Dans notre certification, nous avons utilisé l'option Caduceus –no--fp qui permet d’assimiler les nombres doubles du langage C et les nombres réels de Coq. Une suite logique est donc de prendre en compte les nombres flottants IEEE-754 dans la certification en utilisant la librairie Coq des nombres flottants et le modèle Caduceus des flottants.

L'algèbre linéaire: Dans notre travail, nous avons effectué la certification de deux fonctions des BLAS. Une autre suite à lui donner consiste à certifier d'autres fonctions d'importance capitale dans l'algèbre linéaire (exemple: la factorisation de la bibliothèque LAPACK).

Le type matrice: Nous avons proposé dans Coq une axiomatisation du type matrice en tenant en compte le modèle mémoire de Caduceus. Nous pouvons envisager d'enrichir cette définition par d'autres propriétés sur les matrices et d'autres sur le modèle mémoire.

Le modèle mémoire Caduceus: Ce point découle directement des difficultés que nous avons trouvées dans le raisonnement sur le modèle mémoire Caduceus. Il serait intéressant de considérer ce modèle mais avec un autre type de définition, une définition inductive par exemple.

La méthode Why : La méthode Why permet d'engendrer des obligations de preuves à partir d'un programme annoté. Particulièrement, les annotations comportent les invariants des boucles. Ces invariants ne sont pas faciles à écrire et la méthode ne dispose pas d'un moyen pour aider l'utilisateur de la méthode dans l'écriture des invariants. Nous pouvons envisager une amélioration dans ce sens.

Un autre problème avec Why est la façon dont sont gérées les boucles. Les boucles sont considérées comme des boites noires et seules les invariants sont consultés. Aucune information ne peut être utilisé concernant la suite des modifications d'états mémoire lors du déroulement de la boucle bien que ces informations peuvent être utiles.

Les annexes

Annexe A : Le fichier *dgemm_spec_why.v*

```
(*Why type*) Definition global: Set.
Admitted.
(*Why predicate*) Definition constant_k  (k:Z) := (* CADUCEUS_3 *) k = 1.

(*Why predicate*) Definition constant_lda (lda:Z) := (* CADUCEUS_1*)lda=15.

(*Why predicate*) Definition constant_ldx (ldx:Z):=(* CADUCEUS_2 *)ldx =15.

(*Why predicate*) Definition constant_n  (n:Z) := (* CADUCEUS_4 *) n = 15.

(*why logic*) Definition vect_nul :
  (memory R global) -> (pointer global) -> Prop.

  intros mem y.
  exact (forall i:Z, shift y i #mem = 0%R).
Definied.
```

(****Complémentation de la spécification Caduceus******)**

```
Inductive is_loop_k_aux ( a x y: pointer global) ( i j lda ldx:Z):

memory R global->Z->Prop:=

| elt_k_0: forall k:Z, forall mem, (exists mem0, (shift y (i*ldx+ 0) #mem )
=((shift y (i*ldx+ 0) #mem0) +
((shift a (i * lda+j) #mem0) *(shift x (j * ldx+0)#mem0)))%R)->k=0->
is_loop_k_aux a x y i j lda ldx mem k

| elt_k_s: forall k:Z, forall mem:memory R global, is_loop_k_aux a x y i j
lda ldx  mem k ->

is_loop_k_aux a x y i j lda ldx ( upd mem (shift y (i*ldx+(k+1) ) ) ) ((
shift y (i*ldx+(k+1) )#mem) +

((shift a (i * lda+j) #mem) *(shift x (j * ldx+ (k+1))#mem)))%R)  (k+1).


(*Why logic*) Definition is_loop_k :
(memory R global) -> (pointer global) -> (pointer global) ->
 (pointer global) -> Z -> Z -> Z -> Z -> Z -> Prop.

  intros mem a x y i j k lda ldx.

  exact (is_loop_k_aux  a x y i j lda ldx mem k).

  Defined.

Fixpoint som_prod_aux (a x:pointer global) (mem:memory R global) (i j l lda
ldx:Z) (n:nat) {struct n}:R:=
    match n with
    | 0%nat => (shift a (i*lda+ j) #mem * (shift x (j*ldx+ l) #mem ))%R
    | S n1 =>Rplus (( ( shift a (i*lda+j) #mem )* (shift x (( j)*ldx+ l)
#mem ))%R)
                           (som_prod_aux  a x mem i (j-1) l  lda ldx n1)%R
end.
```

```
Definition sum_prod_j_n (mem:memory R global) (a x:pointer global) (i i1  l
j lda ldx:Z):= match ( j ?= (i1)) with

| Lt => 0%R
| _ => (som_prod_aux a x mem i j l lda ldx  (Zabs_nat (j -( i1))%Z))%R end.


Lemma update_m1_m2: forall (a x y:pointer global), forall (m1 m2:memory R
global), forall (i j  lda ldx:Z),
(exists m, ((shift y (i*ldx+0)) #m1  = ((shift y (i*ldx+0)) #m +
sum_prod_j_n m1 a x  i  0 0 (j)  lda ldx)%R)) ->
(is_loop_k m2 a x y i (j+1) 0 lda ldx) ->(m2 = upd m1 (shift y (i*ldx+0))
( (( shift y (i*ldx+0 )#m1) +
((shift a (i * lda+(j+1)) #m1) *(shift x ((j+1) * ldx+ 0)#m1)))%R)).

Lemma updated_once:
 forall (a x y:pointer global), forall (m m1 m2:memory R global), forall (i
j  lda ldx:Z),
 (forall i1:Z ,  0 <= i1 < i-> ((shift y (i1*ldx+0)) #m1  = ((shift y
(i1*ldx+0)) #m + sum_prod_j_n  m1 a x  i1  0 0 (j)  lda ldx)%R)) ->
 ( shift y (i * ldx + 0) # m2 =  (shift y (i * ldx + 0) # m + sum_prod_j_n
m2 a x  i 0 0 j  lda ldx)%R)->
 ( forall i1 : Z,
   0 <= i1 < i ->
   shift y (i1 * ldx + 0) #m2 =
   (shift y (i1 * ldx + 0) # m+
    sum_prod_j_n m2 a x i1 0 0 j lda ldx)%R).

Axiom not_assigns_intro: forall (a:alloc_table),
           (forall (P: pointer global),
              (forall (m1: memory R global),
                (forall (i:Z),forall (val: R),
         (not_assigns a m1 (upd m1 (shift P i) val) (pset_singleton P))))).


      (******Lemmes sur les prédicats et fonctions******)

Lemma sum_prod_j_n_0: forall (P1: pointer global),
                             forall (P2: pointer global),
                               forall (m1: memory R global),
                                forall  (i  j  m  lda  ldx:Z),  (j<  m)%Z->
(sum_prod_j_n  m1 P1 P2  i m 0 j  lda ldx ) = 0%R.


Lemma sum_prod_j_n_calcul: forall (a x:pointer global), forall (m:memory R
global), forall (i i1 j  lda ldx:Z), (i1<= (j+1))%Z->
      (sum_prod_j_n  m a x i i1 0 (j+1)  lda ldx  )%R =Rplus (Rmult (shift
a (i*lda+ (j+1)) #m)  (shift x ( (j+1)*ldx+ 0)#m))
                         (sum_prod_j_n  m  a x i i1 0 j   lda ldx )%R.



Lemma the_same_som_prod: forall  (a: alloc_table),
                           (*forall (A1:Set), forall (A2:Set),*)
                             forall (P1: pointer global),
                              forall (P2: pointer global),
                               forall (P3: pointer global),
                                forall (m1: memory R global),
                                 forall (m2: memory R global),
```

```
                                  forall (i i1 j l lda ldx:Z), not_assigns
a m1 m2 (pset_singleton P3) ->( sum_prod_j_n m1 P1 P2 i i1 l j  lda ldx  )=
( sum_prod_j_n m2 P1 P2 i i1 l  j  lda ldx  ).
```

(****Le type matrice******)**

```
Definition matrix:Z->Z->Set->Set.
Admitted.

Definition acc_mat: forall (A:Set),forall ( n m :Z),  (matrix n m  A)->Z-
>Z-> A.
 Admitted.
 Implicit Arguments acc_mat.

Definition updte_mat: forall (A:Set) , forall ( n m :Z),( matrix n m  A)
                     ->A->Z->Z-> (matrix n m A).
Admitted.
Implicit Arguments updte_mat.

Axiom acc_updte_mat: forall (A:Set) ,forall (n m:Z),  forall (v:A),
   forall (mat :matrix n m A), forall (i j :Z),(n>0)-> (m>0)->
     (i<=n)  ->  (j <= m) ->(acc_mat (updte_mat mat  v i j) i j)=v.

Definition mat_to_ptr: forall (A:Set),forall (n m :Z),(matrix n m  A)
->(memory A global) -> (pointer global).
Admitted.
Implicit Arguments mat_to_ptr.

Axiom elts_ptr_mat: forall (A:Set), forall(mem: memory A global),
forall ( n m:Z ),  forall (mat:matrix n m A),
      forall ( p: pointer global), (p= mat_to_ptr  mat mem )->
           forall (i j:Z ),(n>0) ->(m>0)->
              (0<=( i) < ( n)) -> (0<= ( j)< ( m))  ->
                  Acc_mat mat i j = shift p ((i)*( m)+(j)) # mem.
```

(****La fonction Coq dgemm_plus******)**

```
Fixpoint  som_prod_mat (n k lda ldx :Z) (a  : matrix (n*n)  lda  R) (x:
matrix  ldx k R) (i  l:Z) (m:nat) {struct m} :R :=
match m with

| 0%nat =>  ((((acc_mat a i  (Z_of_nat m) ))* (acc_mat  x   (Z_of_nat m)
l))%R)
| S m1 =>  (((((acc_mat  a i   (Z_of_nat m)))* (acc_mat  x  (Z_of_nat m)
l))%R)
               + (som_prod_mat   n  k  lda ldx a x i l m1  )%R)%R
end.

 Fixpoint dgemm_plus_aux  (n  k lda ldx :Z) ( a:matrix (n* n) lda  R)  ( x
y : matrix  ldx k R)
              ( n1 m :nat) {struct n1} : (matrix  ldx k R):=
match n1 with

| 0%nat => (updte_mat y (som_prod_mat n k lda ldx a x (Z_of_nat n1) 0  m
)%R  (Z_of_nat n1) 0  )
| S n2 => let yt := (updte_mat  y  ((som_prod_mat n k lda ldx   a x
(Z_of_nat n1) 0  m  ))%R  (Z_of_nat n1) 0 ) in
  dgemm_plus_aux  n k lda ldx  a x yt n2  m
```

```
end.

  Definition dgemm_plus (n k lda ldx m:Z ) ( a:matrix (n* n) lda  R)
                           (x y : matrix  ldx k R) : (matrix  ldx k R).

  intros n k lda ldx m  a x y .
  exact (dgemm_plus_aux  n k lda ldx a x y (Zabs_nat (n-1)) (Zabs_nat m)).
  Defined.
Lemma som_prod_mat_ptr: forall (mem: memory R global),
                 forall (p1 p2: pointer global), forall (n k lda ldx : Z),
                      forall (a: matrix (n*n) lda R),
                   forall (x: matrix  ldx k R), (p1 = mat_to_ptr a mem) ->
(p2 = mat_to_ptr x mem) ->  forall (i :Z),(n>0)-> 0<=i<n ->
(som_prod_mat n k lda ldx a x i 0 (Zabs_nat (n-1)))%R =
                          (sum_prod_j_n  mem p1 p2 i 0 0 (n-1)  lda ldx)%R.


Lemma dgemm_plus_elts: forall ( n k lda ldx :Z),  forall (a: matrix (n*n)
lda R),forall (x y yt : matrix ldx k R),forall i: Z, 0<=i<n->
  (yt = dgemm_plus n k lda ldx n a x y) -> ((acc_mat  yt i 0) =
som_prod_mat n k lda ldx a x i 0 (Zabs_nat (n-1))).
  Admitted.
```

(******Le prédicat DGEMM_is_dgemm******)

```
(*Why logic*) Definition DGEMM_is_dgemm :
  (memory R global) -> (pointer global) -> (pointer global)->
                   (pointer global) -> Z -> Z -> Z -> Z -> Z -> Prop.

intros mem a x y  n m k lda ldx.

exact ( forall (Am: matrix (n*n) lda R), forall(Xm Ym Yt: matrix ldx k R ),
(a = mat_to_ptr  Am mem )-> (x = mat_to_ptr  Xm mem )
  ->(Yt =( dgemm_plus n k lda ldx m Am Xm Ym)) ->
forall( p: pointer global),forall (i:Z), (p= mat_to_ptr Yt mem) ->
(  0<=i<n)  ->(shift  y  (i*ldx  +  0)  #mem)  =  (shift  p  (i*ldx  +0)#mem)).
Defined.
```

Annexe B : Le fichier *dtrsm_spec_why.v*

```
(*Why type*) Definition global: Set.
Admitted.

(*Why predicate*) Definition constant_k  (k:Z) := (* CADUCEUS_3 *) k = 1.

(*Why predicate*) Definition constant_lda  (lda:Z):=(* CADUCEUS_1*) lda=15.

(*Why predicate*)Definition constant_ldx (ldx:Z) :=(* CADUCEUS_2 *)ldx =15.

(*Why predicate*) Definition constant_n  (n:Z) := (* CADUCEUS_4 *) n = 15.
```

(****Complémentation de la spécification Caduceus******)**

```
(*Why logic*) Definition diag_un : (memory R global) -> (pointer global) ->
Z -> Z -> Prop.
intros mem a n lda.
exact (forall (i:Z), 0<=i<n -> (acc mem (shift a (i*lda+i))) = 1%R).
Defined.

(*Why logic*) Definition triang_sup : (memory R global) -> (pointer global)
-> Z -> Z -> Prop.
intros mem a n lda.
exact (forall (i j:Z), 0<=i<n -> (j < i) -> (acc mem (shift a (i*lda+j))) =
0%R).
Defined.

(*Why logic*) Definition diff_ptr:
   (pointer global) -> (pointer global) -> Prop.
intros a x.
exact (base_addr a <> base_addr x).
Defined.

Inductive is_loop_k_aux_2 ( a x y: pointer global) ( i j lda ldx:Z):

memory R global->Z->Prop:=

| elt_k_0_2: forall k:Z, forall mem, (exists mem0, (shift y (i*ldx+ 0) #mem
) =((shift y (i*ldx+ 0) #mem0) -
((shift a (i * lda+j) #mem0) *(shift x (j * ldx+0)#mem0)))%R)->k=0->
is_loop_k_aux_2 a x y i j lda ldx mem k

| elt_k_s_2: forall k:Z, forall mem:memory R global, is_loop_k_aux_2 a x y
i j lda ldx  mem k ->

is_loop_k_aux_2 a x y i j lda ldx ( upd mem (shift y (i*ldx+(k+1) ) ) ) ((
shift y (i*ldx+(k+1) )#mem) -

((shift a (i * lda+j) #mem) *(shift x (j * ldx+ (k+1))#mem)))%R)  (k+1).
```

```
(*Why logic*) Definition is_loop_k_2 :
  (memory R global) -> (pointer global) -> (pointer global)
  -> (pointer global) -> Z -> Z -> Z -> Z -> Z -> Prop.

  intros mem a x y i j k lda ldx.

  exact (is_loop_k_aux_2  a x y i j lda ldx mem k).

  Defined.

Lemma update_m1_m2_2: forall (a x :pointer global),
              forall (m1 m2:memory R global), forall (i j  lda ldx:Z),
(exists  m,  ((shift  x  (i*ldx+0))  #m1    =  ((shift  x  (i*ldx+0))  #m  -
sum_prod_j_n  m1 a x  i (i+1) 0  (j)  lda ldx)%R)) ->
(is_loop_k_2 m2 a x x i (j+1) 0 lda ldx) ->(m2 = upd m1 (shift x (i*ldx+0))
( (( shift x (i*ldx+0 )#m1) -
((shift a (i * lda+(j+1)) #m1) *(shift x ((j+1) * ldx+ 0)#m1)))%R)).

Lemma updated_once_2:
 forall (a x y:pointer global), forall (m m1 m2:memory R global), forall (i
j  lda ldx n:Z),
 (forall i1  :Z ,  i < i1 <=n-2 -> ((shift x (i1*ldx+0)) #m1  = ((shift x
(i1*ldx+0)) #m - sum_prod_j_n  m1 a x  i1  (i1+1)  0 (j)  lda ldx)%R)) ->
 ( shift x (i * ldx + 0) # m2 =  (shift x (i * ldx + 0) # m - sum_prod_j_n
m2 a x  i (i+1) 0 j  lda ldx)%R)->
 ( forall i1 : Z,
   i< i1 <= n-2  ->
   shift x (i1 * ldx + 0) #m2 =
   (shift x (i1 * ldx + 0) # m-
    sum_prod_j_n m2 a x i1 (i1+1)  0 j lda ldx)%R).

Axiom not_assigns_intro_2: forall (a:alloc_table),
                             (forall (P: pointer global),
                               (forall (m1: memory R global),
                                 (forall (i:Z),forall (val: R),
 (not_assigns  a  m1  (upd  m1  (shift  P  i)  val)  (pset_singleton  (shift  P
i))))))).
```

(****Lemmes sur les prédicats et fonctions******)**

Voir l'annexe A.

(****Le type matrice******)**

Voir l'annexe A.

(****Le prédicat property******)**

```
(*Why logic*) Definition property:
  (memory R global) -> (pointer global) -> (pointer global) -> Z -> Z -> Z
  -> Z -> Prop.
intros mem a x  n  k lda ldx.
```

```
exact ( forall (Am: matrix (n*n) lda R), forall (Xm X1m Yt: matrix  ldx k R
), (a = mat_to_ptr  Am mem )-> (x = mat_to_ptr  Xm mem )
  -> (X1m = (dgemm_plus n k lda ldx n Am Xm Yt))
  -> forall (x1: pointer global), x1 = mat_to_ptr X1m mem
        -> (exists mem0, (forall (i1: Z),
                  (0 <= i1 /\ i1 <= (n - 2) ->
                   (eq (acc mem (shift x (i1 * ldx + 0))) ( Rminus
                    (acc mem0 (shift x (i1 * ldx + 0))) (sum_prod_j_n mem a
x i1 (i1 + 1) 0 (n - 1) lda ldx))))
-> (forall (i1: Z), 0<= i1 <= (n-2) ->
((acc mem (shift x1 (i1 * ldx + 0))) = (acc mem0 (shift x(i1 * ldx +
0))))))))).

Defined.

Lemma som_prod_triang_sup_0:
 forall (a x:pointer global), forall (i lda ldx n j:Z),
                 forall (mem: memory  R global),
              (0<= i< n) -> (j < i) -> (triang_sup mem a n lda) ->
(sum_prod_j_n mem a x i 0 0 j lda ldx = 0%R).

Lemma som_pro_j_eq_n:
 forall (mem : memory R global), forall (a x: pointer global), forall (i
lda ldx :Z), (sum_prod_j_n mem a x  i i 0 i lda ldx)%R =
                 (shift a (i*lda+ i) #mem * (shift x (i*ldx+ 0) #mem ))%R.

Lemma som_prod_split:
 forall (a x:pointer global), forall (mem: memory R global),
                   forall (i b c lda ldx:Z), (b < c)  -> ( b <= i <= c) ->
((sum_prod_j_n mem a x i b 0 c lda ldx)%R =
         ((sum_prod_j_n mem a x i b 0 (i-1) lda ldx) +
                    (sum_prod_j_n mem a x i i 0 i lda ldx)+
                         (sum_prod_j_n mem a x i (i+1) 0 c lda ldx))%R).
Admitted.
```

Bibliographie

Bibliographie

[01] The Coq proof assistant reference manual, july 2007. http://coq.inria.fr.

[02] Yves BERTOT, Pierre CASTERAN. Interactive Theorem Proving and Program Development, Coq'Art: the calculus of inductive constructions. Springer-Verlag , 2004.

[03] Mohamed CHAABANI. Spécification, preuve et extraction automatique des programmes en Coq. Cas: L'algorithme $\lambda c\beta^{+}$ - réduction. Mémoire de magister, Université M'hamed BOUGARA de Boumerdes. 2003.

[04] Antonia BALAA. Fonctions récursives générales dans le calcul des constructions. Thèse de Doctorat, Université de Nice-Sophia Antipolis, 2002.

[05] Jean-Christophe FILLIATRE. The Why verification tool reference manual. http://why.lri.fr/

[06] Jean-Christophe FILLIATRE et Claude MARCHE. Multi-prover verification of C programs. In Sixth International Conference of Formal Engineering Methods, Seattle, 2004.

[07] Jean-Christophe FILLIATRE. Why: a multi-language multi-prover verification condition generator. Rapport de recherche 1366, LRI, Université Paris Sud, 2003.

[08] Jean-Christophe FILLIATRE. Preuve de programmes impératifs en théorie des types. Thèse de Doctorat, université de Paris XI, Orsay, 1999.

[09] June ANDRONICK-LIEGE: Modélisation et vérification formelle de systèmes embarquée dans les cartes à microprocesseurs. Plate-forme Java Card et système d'exploitation. Thèse de Doctorat, université de Paris XI, Orsay, 2006.

[10] Philip WADLER: Monads for functional programming. In Proceedings of the Marktoberdorf Summer School on Program Design Calculi. August 1992.

[11] Jean-Pierre TALPIN et Pierre JOUVELOT. The type and effect discipline. Information and Computation. 1994.

[12] Micaela MAYERO. Formalisation et automatisation de preuves en analyse réelle et numérique. Thèse de Doctorat, Université Paris 6, 2001.

[13] Sylvain BARO. Conception et implémentation d'un système d'aide à la spécification et à la preuve de programmes ML. Thèse de Doctorat de l'université Paris 7, juillet 2003.

[14] J.GUTTAG AND J.HORNING. Larch: Languages and Tools for Formal Specification. Springer-Verlag, 1993.

[15] Catherine PARENT. Synthèse de preuves de programmes dans le Calcul des Constructions Inductives. Thèse de Doctorat. Ecole Normale Supérieure de Lyon, 1995.

[16] Micaela Mayero. Using Theorem Proving for Numerical Analysis. Correctness Proof of an Automatic Differentiation Algorithm. In Proceedings of TPHOLs2002, volume 2410. Springer-Verlag LNCS.

[17] Chris OKASAKI. From fast exponentiation to square matrices: an adventure in types. International Conference on Functional Programming, 1999.

[18] Nicolas MAGAUD. Programming with dependent types in Coq: a study of square matrices. 2005. Unpublished. http://dpt-info.u-strasbg.fr/~magaud/UNSW/matricesinCoq.pdf

[19] Milad NIQUI. Formalising Exact Arithmetic: Representations, Algorithms and Proofs. PhD thesis, Institute for Computing and Information Sciences of Radboud University of Nijmegen 2004.

[20] Yves BERTOT. Calcul de formules affines et de séries entières en arithmétique exacte avec types co-inductifs. In Thérèse Hardin et Luc Moreau, editor, Journées francophones des langages applicatifs. INRIA, Janvier 2006.

[21] Yann REGIS-GIANAS. Des types aux assertions logiques : Preuve automatique ou assistée de propriétés sur les programmes fonctionnels. Thèse de Doctorat, Université Paris 7 – Denis Diderot, 2007.

[22] J. Christian ATTIOGBÉ. Contributions aux approches formelles de développement de logiciels Intégration de méthodes formelles et analyse multifacette, Rapport scientifique pour une Habilitation à Diriger des Recherches, Université de Nantes,2007.

[23] Marianne SIMONOT: Preuve de programmes impératifs. Notes de cours Master Stic parcours logiciels surs. Valeur C Construction Rigoureuse de Logiciel. CNAM, 2004.

[24] Yamine AIT-AMEUR. Spécifications formelles. Cours de Post-Graduation. Université M'hamed BOUGARA, Boumerdes, 2007.

[25] Richard BORNAT. Proving pointer programs in Hoare Logic. In R. Backhouse and J. Oliveira, editors, Mathematics of Program Construction (MPC 2000), volume 1837 of LNCS, pages 102–126. Springer, 2000.

[26] F. MEHTA and T. NIPKOW. Proving pointer programs in higher-order logic. Information and Computation, Volume 199. 2005.

[27] Michael NORRISH. C formalised in HOL. Technical Report. University of Cambridge. Computer Laboratory.December 1998.

[28] Nicolas MAGAUD. Changement de représnetation des données dans le calcul des constructions. Thèse doctorat. Université de Nice-Sophia Antipolis, 2003.

[29] M. BEN-ARI. Mathematical logic for computer science . Springer, second edition, 2001.

[30] Sylvie BOLDO et Jean-Christophe FILLIATRE. Formal Verification of Floating-Point Programs. In the 18TH Symposium On Computer Arithmetic. Montpellier, France, juin 2007.

[31] John C. REYNOLDS. An overview of separation logic. Computer ScienceDepartement. Carnegie Mellon University.2002.

[32] Norbert SCHIRMER. Verification of Sequential Imperative Programs in Isabelle/HOL. Thèse de doctorat. Université de MUNICH, 2005.

[33] C.A.R. HOARE: An axiomatic basis for computer programming. Communication of the ACM, 12: 567-580, 583, 1969.

[34] Marie-Claude GAUDEL, Michel LEMOINE, Bruno MARRE, Françoise SCHLIENGER et Gilles BERNOT. Précis du génie logiciel. Masson, 1996.

[35] Nikolaj POPOV. Verification using Weakest Precondition strategy. Contributed talk at Computer Aided Verification of Information Systems (CAVIS-04). Timisoara, Romania., 2003.

[36] Guillaume MELQUIOND. De l'arithmétique d'intervalles à la certification de programmes. Thèse de doctorat, École Normale Supérieure de Lyon, 2006 ;

[37] Francisco José CHÁVES ALONSO. Utilisation et certification de l'arithmétique d'intervalles dans un assistant de preuves. Thèse de Doctorat. École Normale Supérieure de Lyon 2007.

[38] Valérie MÉNISSIER-MORAIN. Arithmétique exacte : Conception, algorithmique et performances d'une implémentation informatique en précision arbitraire. Thèse de Doctorat. Université Paris VII, 1994

[39] Marc DAUMAS, Laurence RIDEAU et Laurent THERY: A generic library of floating-point numbers and its application to exact computing. In 14th International Conference on Theorem Proving in Higher Order Logics, pages 169–184, Edinburgh, Scotland, 2001.

[40] Sylvie BOLDO. Preuves formelles en arithmétiques à virgule flottante. Thèse de Doctorat. Ecole Normale Supérieure de Lyon, 2004.

[41] David DEFOUR. Fonctions élémentaires: algorithmes et implémentation efficaces pour l'arrondi correct en double précision. Thèse de doctorat. Ecole Normale Supérieure de Lyon, 2003.

[42] John HARRISON. Theorem proving with real numbers. Springer-Verlag. 1998.

[43] Renaud RIOBOO. Quelques aspects du calcul exact avec les nombres réels. Thèse de doctorat. Université Paris 6, 1991.

[44] Ramon Edgar MOORE. Interval arithmetic and automatic error analysis in digital computing. Thèse Doctorat. Université de Stanford, 1962.

[45] John HARRISON. Floating point verification in HOL light : the exponential function.Technical Report 428, University of Cambridge Computer Laboratory, 1997.

[46] Paul S. MINER. Defining the IEEE-854 floating-point standard in PVS. Rapport technique 110167, NASA Langley Research Center, 1995.

[47] Christine PAULIN-MOHRING, Benjamin WERNER, Bruno BARRAS, Hugo HERBELIN, Jean-Christophe FILLIATRE et Claude MARCHE: Cours Assistants de preuves, Master parisien de recherche en informatique, 2008.

[48] David von Oheimb. Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic. Thèse de Doctorat.Université de Munich, 2001.

[49] José Luis Freire Nistal, Enrique Freire Branas, Antonio Blanco Ferro, and David Cabrero Souto. On the Representation of Imperative Programs in a Logical Framework. Springer-Verlag, 2007.

[50] Edsger W. DIJKSTRA: Notes on structured programming. In O .- J. Dahl, E.W. Dijkstra and C.A.R. Hoare, editors, Structured Programming. Academic Press, 1972.

[51] Jeannette M. WING: A specifier's introduction to formal methods. Carnegie Mellon University, 1990.

[52] John RUSHBY. Formal methods and their role in the certification of critical systems. Technical Report SRI-CSL-95-01, Computer Science Laboratory, SRI International ,1995.

[53] Olivier PONSINI, Carine FEDELE et Emmanuel KOUNALIS. Rewritting of imperative programs into logical equations. Published by Elsevier North-Holland, 2004.

[54] Axel Van LAMSWEERDE: Formal specification: a roadmap. In the Future of Software Engineering, A. Finkelstein, ACM Press, 2000.

[55] Emil SEKERINSKI. Program development by refinement: case studies using the B method. Springer, 1999

[56] Guillaume DUFAY: Vérification formelle de la plate-forme JavaCard. Thèse de Doctorat, Université de Nice- Sophia Antipolis, 2003.

[57] Simao Melo DE SOUZA: Outils et techniques pour la vérification formelle de la plate-forme JavaCard. Thèse de Doctorat, 2003.

[58] Gary T. LEAVENS et al. JML: Notations and tools supporting detailed design in Java. In OOPSLA 2000 Companion. ACM 2000.

[59] Freek WIEDIJK. Comparing mathematical provers. In Andrea Asperti, Bruno Buchberger, and James Davenport, editors, Proceedings of Mathematical Knowledge Management (MKM), 2003.

[60] Jean-François COUCHOT. Vérification d'invariants de systèmes paramétrés par superposition. Thèse de doctorat. Université de Franche-Comté, 2006.

[61] HOL. http://www.cl.cam.ac.uk/Research/HVG/HOL/HOL.html.

[62] http://www.cl.cam.ac.uk/research/hvg/Isabelle/.

[63] The Simplify decision procedure. http://research.compaq.com/SRC/esc/ simplify/.

[64] The haRVey decision procedure. http://www.loria.fr/_ranise/haRVey/.

[65] http://compcert.inria.fr/

[66] The PVS system: http://pvs.csl.sri.com/.

[67] Clément PERNET. Algèbre linéaire exacte efficace : le calcul du polynôme caractéristique. Thèse de doctorat. Université JOSEPH FOURIER, 2006.

[68] Jocelyne ERHEL Nabil NASSIF et Bernard PHILIPPE. Calcul matriciel et systèmes linéaires. Mémoire DEA Informatique et modélisation. Université de Liban, 2004.

[69] Yoann LE BARS. Implémentation de la méthode CESTAC dans la bibliothèque BLAS. Mémoire de Master 2 mathématiques de la modélisation. Université Pierre et Marie Curie —Paris 6, 2005.

[70] Rob H. BISSELING. Parallel scientific computation: a structured approach. Oxford University Press, 2004.

[71] Frédéric DESPREZ. Contribution à l'algorithmique parallèle. Calcul numérique : des bibliothèques aux environnements de metacomputing. Thèse d'habilitation diriger des recherches. Université Claude Bernard de LYON, 2001.

[72] Pascal GIORGI. Arithmétique et algorithmique en algèbre linéaire exacte pour la bibliothèque LINBOX. Thèse de Doctorat. Ecole Normale Supérieure de Lyon, 2004.

[73] Basic Linear Algebra Subprograms. A quick Reference guide. University of Tennessee. Oak Ridge National Laboratory. Numerical Algorithms Group Ltd, May 1997.

[74] Le site de la bibliothèque BLAS. http://www.netlib.org/blas/index.html

[75] http://www.netlib.org/lapack/

[76] J.R. HINDLEY and J.P. SELDIN. Introduction to lambda-calculus and combinators. Cambridge University Press, 2nd Edition, 2007.

[77] http:// www.lipn.univ-paris13.fr/CerPAN/

[78] Sylvie BOLDO, Jean-Christophe FILLIATRE et Guillaume MELQUIOND. Combining Coq and Gappa for certifying floating-poit programs. To appear In 16th Symposium on the Integration of Symbolic Computation and Mechanised Reasoning. LNCS/LNAI, July 2009.

[79] Hanne GOTTLIEBSEN. Trancendental functions and continuity checking in PVS. In Proc. TPHOL. Springer LNCS, September 2000.

[80] http://fost.saclay.inria.fr/index.html

[81] www.theorema.org