UNIVERSITÉ DE NAMUR

Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche

researchportal.unamur.be

University of Namur

THESIS / THÈSE

MASTER EN SCIENCES INFORMATIQUES

Analyse statique par calcul de point fixe de systèmes d'équations

Maes, Stéphane

Award date:
1992

Awarding institution:
Universite de Namur

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 02. May. 2026

Facultés Universitaires Notre-Dame de la Paix à Namur
Institut d'informatique

Analyse statique par calcul de point fixe de systèmes d'équations

Promoteur

Professeur Baudouin Le Charlier

Mémoire présenté par

Stéphane Maes

en vue de l'obtention du grade de
Licencié et Maître en informatique

Année académique 1991-1992

rue Grandgagnage, 21 B-5000 Namur Belgique

Analyse statique
par calcul de point fixe
de systèmes d'équations

Stéphane MAES

Abstract

Considering a program is able to compute all the solutions of a category of problems (called domain of application), we could say that the more restricted the category of problems is, the more fitted and optimal the code of programming is.
The idea of the thesis is the following : the available information are used into a system of equations within variables get a value corresponding to a category of problems. An equation defines either a "single" variable (these takes only one value), either a "function" variable (these is a table of correspondences). Then the software is computing the best feasible approximation of "single" variables by means of fixpoint. The used algorithm try to avoid needless calculations by performing a graph of dependances.

Résumé

L'analyse statique de programmes permet souvent de mieux cerner le domaine d'un programme et autorise ainsi à en spécialiser le code. Ce travail propose d'organiser les informations disponibles en système d'équations et fournit un solveur par calcul de point fixe. L'algorithme tente d'éviter les calculs inutiles par construction d'un graphe des dépendances.

Remerciements

Ce travail, ces années d'études n'auraient pu être menés sans le soutien de Christine ; je lui dois beaucoup, beaucoup.

Merci à Yvan pour son aide ; merci à Antonin, Anaël et Pierre pour leur patience ; attention, on va s'y remettre !

J'aimerais saluer, une personne que j'ai eu grand plaisir à apprécier et qui restera toujours un modèle, Merci Baudouin.

Arthur est né au beau milieu de la rédaction de ce travail.
Je termine vite, pour le rejoindre.

Les langages modèlent notre pensée et déterminent les objets possibles de celle-ci. B.L.WHORF

PLAN

A. Présentation du travail p. 9

B. Présentation du logiciel p.12

C. Applications p.34

D. Sémantique dénotationnelle du programme p.42

E. Principaux algorithmes p.65

F. Extensions possibles p.78

Bibliographie p.79

ANNEXE : Fondements théoriques - théorie du point fixe p.80

Note : il n'a pas été jugé utile de présenter le listing du programme ; il est très long et peu abordable en première lecture.

PRESENTATION DU TRAVAIL

en bref :

Au plus un problème est général, au plus il est simple à résoudre ; mais au plus un algorithme est spécialisé au plus il est efficace.
Si l'on considère qu'un programme informatique dénote toutes les solutions de la classe de problèmes qu'est son domaine d'application, au plus la classe est restreinte, au plus le code peut être adapté, optimisé ; c'est aussi valable pour un "sous-programme" : on a tout intérêt à connaître avec précision la plus petite borne supérieure des objets du sous-programme.

Ce travail va proposer d'organiser les informations disponibles en système d'équations où les variables ont une valeur dénotant une classe de problèmes. Une équation définira soit une variable 'simple' : qui n'a qu'une valeur, soit une variable 'fonction' : qui est une table de correspondances. Le logiciel va alors rechercher la meilleure approximation possible de variables 'simples' par calcul de point fixe. L'algorithme utilisé tente d'épargner le plus de calculs inutiles en construisant un graphe de dépendances [1].

Le logiciel se veut générique et modulaire c'est-à-dire qu'il peut sans trop d'efforts être adapté à de nouvelles classes de problèmes ; mais la création du système d'équations permettant de demander ce qu'on appellera une "interprétation abstraite" reste dans la plupart des cas une tâche compliquée et il restera à modifier (ou non) les codes de son programme. Entre-temps, ce logiciel pourra être utilisé comme 'didacticiel de point fixe'.

Le logiciel fourni comporte un analyseur lexical, un analyseur syntaxique, un analyseur sémantique (de types), un processeur d'approximations (calcul de point fixe) ainsi qu'une batterie de domaines et opérateurs prédéfinis sur ces domaines, son interface d'exploitation est très réduite et de nombreuses optimisations sont possibles. Il n'a pas été jugé utile de porter un soin particulier à ces deux derniers aspects.

---

1 : L'algorithme est emprunté à A Universal Top-Down Fixpoint Algorithm Baudouin Le Charlier et Pascal Van Hentenryck.

Exposons maintenant plus en détails les objectifs du travail.

interprétation abstraite

Soit la ligne de programme

```
x := if b<0 then b*b else b*c*d*factorielle(8437)
```

Il est clair que si b est toujours <0, la conditionnelle est superflue et la variable x aura toujours une valeur positive ; il est clair aussi que si b>=0 alors si b ou c ou d est nul, le résultat sera 0 et il est inutile de calculer la factorielle de 8437 (ouf!).

Analyser ainsi un programme, c'est effectuer une interprétation abstraite ; 'interprétation' car on propose des solutions conformes aux contraintes ; 'abstraite' car on ne s'intéresse pas aux valeurs concrètes des variables mais à des classes de valeurs recouvrant certaines propriétés jugées intéressantes ; en l'occurrence les classes de valeurs dénotées par <0 =0 >0.

domaines abstraits et domaines concrets

Les classes des valeurs possibles d'une variable de programme sont complétées par deux éléments l'un désigné par ⊥ qui signifiera "on ne sait rien", l'autre par ⊤ qui signifiera "c'est tout à fait indécidable".

Comme on redéfinit les domaines des variables, on doit redéfinir également les fonctions ou opérateurs de telle sorte qu'ils 'travaillent' dans ces nouveaux domaines. Voici deux exemples :

a*b doit donner :
- si (a=⊥ ou b=⊥) et a≠⊤ et a≠⊤ alors ⊥
- si a<0 et b<0 alors >0
- si a<0 et b>0 alors <0
- si a=0 ou b=0 alors =0
- si a>0 et b<0 alors <0
- si a>0 et b>0 alors >0
- si a=⊤ ou b=⊤ alors ⊤

a+b doit donner :
- si (a=⊥ ou b=⊥) et a≠⊤ et a≠⊤ alors ⊥
- si a<0 et b<0 alors <0
- si a<0 et b=0 alors <0
- si a<0 et b>0 alors ⊤
- si a=0 et b<0 alors <0
- si a=0 et b=0 alors =0
- si a=0 et b>0 alors >0
- si a>0 et b<0 alors ⊤
- si a>0 et b=0 alors >0
- si a>0 et b>0 alors >0
- si a=⊤ ou b=⊤ alors ⊤

On voit qu'il y a des "pertes d'informations" ; par exemple, lorsqu'il est écrit "si a<0 et b>0 alors ⊤". En effet, concrètement, seule l'une des trois possibilités <0 =0 >0 sera effective mais on n'a pas la possibilité d'en décider. L'interprétation abstraite donnera un résultat qu'il faut considérer comme "minimal" : concrètement, les données seront probalement mieux définies :

dans le domaine abstrait : a+b avec a<0 et b>0 donnera ⊤
alors que dans le domaine concret : -5+3 donnerait <0 !

Soit encore, la fonction factorielle que l'on pourrait définir par

fact(x) $\triangleq$ si x=0 alors 1 sinon x*fact(x-1)

Cette fonction retournera-t-elle un résultat $\bot$, <0, =0, >0, ou $\top$ si son argument est >0 ?
Effectuons une exécution dans le domaine abstrait {$\bot$,<0,=0,>0,$\top$} en assignant à x la valeur >0 :

fact(>0)=(>0)*fact((>0)-(>0))

(>0)-(>0) donne $\top$, et il faudra encore calculer fact($\top$) qui ne pourra donner que $\top$; donc, finalement fact(>0)=$\top$.
Ce résultat n'est pas glorieux ; puisque nous savons que fact(>0) donne toujours un résultat >0 !

L'interprétation abstraite est une évaluation des propriétés d'un programme, effectuée en substituant aux domaines concrets des variables des domaines abstraits qui dénotent certaines propriétés jugées intéressantes. L'évaluation est effectuée par une simple exécution du programme ainsi modifié et énonce une solution minimale.

systèmes d'équations

Notre programme évalue des systèmes d'équations ; avant d'exposer comment une évaluation est effectuée et quelles sont les contraintes, exposons pourquoi nous avons choisi cette notation.
Une équation est une formulation qui assigne à une variable une valeur définie par une expression qui articule des fonctions, des opérateurs et d'autres variables. Un système d'équations est une succession d'équations telle que toute variable est définie par une et une seule expression.
Ce langage de description est bien connu ; il a été choisi, d'abord parce que la plupart des langages de programmation procéduraux et leurs programmes peuvent être traduits sous cette forme (la forme 'fonctionnelle'), ensuite parce qu'il est très approprié aux définitions croisées ou récursives.
Moyennant un effort de traduction, il y a moyen d'exposer en système d'équations, et les propriétés du programme à analyser, et les renseignements abstraits à fournir à l'évaluateur (se restreindre à une sous-classe de problèmes). Une technique générale sera proposée en ce sens.

Les domaines abstraits les plus usuels et certains opérateurs sur ces domaines sont fournis par le programme mais il est possible de les créer. Cette dernière éventualité est d'ailleurs essentielle puisque l'interprétation abstraite est avant tout une redéfinition de types et opérateurs.

monotonie des fonctions

Pour démontrer la correction d'une interprétation abstraite, on va devoir démontrer que tout calcul effectué dans les domaines abstraits est d'application dans les domaines concrets ; c'est-à-dire que si par exemple (<0)+(<0) donne (<0) toute concrétisation de (<0)+(<0) donne un résultat qui appartient à la classe (<0). Mais ce n'est pas tout, intuitivement, en reprenant l'exemple ci-dessus du domaine abstrait {$\bot$,<0,=0,>0,$\top$} il faut que si une évaluation donne comme résultat $\top$ toute évaluation qui suivra et porte sur la même variable ne pourra donner un résultat autre que $\top$ puisque $\top$ signifie "tout-à-fait indécidable" ; par contre un résultat $\bot$ pourra être amélioré ; semblablement, si l'on a pu déterminer qu'une variable est <0 on ne peut plus revenir en arrière, ni dire qu'elle est =0 ou >0 ; par contre certaines situations pourront faire en sorte qu'elle deviendra $\top$: ce seront les situations conflictuelles telle que a:=b/0.

Chaque domaine de l'interprétation abstraite va devoir être muni d'une relation d'ordre partiel $\preceq$ conduisant de $\perp$ à $\top$, et chaque opération devra au moins préserver cet ordre : on dira que les fonctions doivent être monotones.

Petite parenthèse pour que la notion de monotonie soit bien comprise : on peut dire à un enfant que tous les oiseaux volent puis - plus tard - lui expliquer que les pingouins ne volent pas ni les oiseaux morts ; pas à notre programme ! Cette contrainte sera essentielle pour chaque variable de notre système d'équations.

Le principe de monotonie va imposer que les écritures :

f(x) := si x=$\top$ then $\perp$ else $\top$.
a := if a==$\perp$ then $\top$ else $\perp$.

sont interdites.
Par contre "a:=a+1;" est autorisé ; mais ça posera quelques problèmes que l'on traitera plus loin.
L'écriture "a:=b/c;" ne pose pas de problème si l'on définit x/0 par $\top$.
Par contre, comment considérer "a:=f(b/0);f(x):=2;" ; que faire de f($\perp$) ou f($\top$) ? plusieurs solutions sont envisageables, on ne pourra accepter que les solutions monotones ; par exemple :

- f $\triangleq$ $\forall$x f(x)=2
- f $\triangleq$ si x=$\top$$\vee$x=$\perp$ alors x sinon 2

On conçoit maintenant que lors d'une interprétation abstraite, on initialise toutes les variables du programme à $\perp$ puis l'on tente de résoudre chaque équation afin d'améliorer chaque valeur abstraite calculée. On acceptera qu'une variable utilise la valeur d'une autre variable (ou d'elle-même), de même pour une fonction. On devra dresser une carte des dépendances et recommencer les calculs des variables dépendantes. Mais combien de fois ?

définitions récursives

Le problème posé est celui des définitions récursives : jusqu'à présent, notre relation d'ordre sur les éléments du domaine ne permettait de construire que des chaînes strictement croissantes finies ce qui permet d'assurer la terminaison des approximations. Par contre, si l'on admet des domaines définis récursivement tels que :

Liste_d'entiers ::= []   (la liste vide)
Liste_d'entiers ::= [x|y] avec x est un entier
                                        et y $\in$ Liste_d'entiers.

on risque d'approximer sans fin.
Remarque : il faut une relation d'ordre ! ; prenons par exemple :

$\forall$ a,b $\in$ Liste_d'entiers
  a$\preceq$b $\Leftrightarrow$ a=[] $\vee$ (a=[x|y] $\wedge$ b=[x|z] $\wedge$ y$\preceq$z)

Chaque objet d'un domaine ainsi construit peut déjà être défini comme la limite d'une suite croissante d'objets dans le domaine.
Inversément, pour qu'un domaine soit valide, il faudra que toute suite croissante (même infinie) d'objets du domaine définisse un et un seul objet du domaine ; on dira que le domaine est inductif.

Le domaine des "nombres naturels" n'est pas inductif pour la relation $\leq$ car on peut définir une suite croissante ($x_{i+1}=x_i+1$) qui définit un objet (∞) qui n'est pas un nombre naturel. La théorie mathématique des limites nous donne beaucoup d'exemples de ce type : PI est défini par une suite croissante dans les nombres rationnels mais n'est pas rationnel.

Si l'on écrit "x:=[1,2,3]" avec x du domaine Liste_d'entiers, il faudra en fait considérer que x est défini par la suite croissante

[] ≼ [1|[]] ≼ [1|[2|[]]] ≼ [1|[2|[3|[]]]] ≼ ...

Notre programme construira x en avançant de gauche à droite ; on ne reviendra pas en arrière pour autant que la monotonie soit respectée ; mais on peut chercher indéfiniment : x:=[1|x] engendra une construction infinie ! On retombe dans le problème x:=x+1 avec x du domaine des nombres naturels ; quelque chose ne va pas !

Pour que cela aille, il faut inverser le problème : soit x, un élément de Liste_d'entiers, cet élément définit un espace inductif : l'ensemble de tous les éléments de Liste_d'entiers qui lui sont "plus petits" = {y|y≼x} et x pourra être défini comme étant le plus grand de tous ceux-là. C'est trivial et pourtant essentiel : toute équation doit être la définition d'un ensemble d'objets qui peuvent former une chaîne (finie ou infinie) et telle que la solution de l'équation est le plus grand élément de cette chaîne.

La notion d'espace inductif n'est en fait intéressante qu'à propos d'ensembles infinis ce qui n'est pas le cas de l'ensemble {y|y≼x}.
Il a déjà été signalé que dans nos systèmes d'équations on pourra avoir une variable qui est une fonction. Soit f, une fonction de A --> B, le domaine de cette variable est le domaine des fonctions de A dans B, la relation d'ordre est que si f et g sont des fonctions de A dans B, f≼g si et seulement si pour tout x tel que f(x) est défini on a g(x)=f(x). Dans ce domaine, on pourra construire une suite croissante qui définit f avec la particularité que cette suite peut comporter une infinité d'objets distincts (si A comporte une infinité d'objets distincts comme les ensembles définis récursivement). Pour considérer que la suite définit f sans ambiguité il faut passer à la limite ; ce qui va être exprimé au travers de la notion de continuité.

continuité des fonctions

Pour définir une fonction dont le domaine de validité est infini, le plus commode, est d'exploiter la récursivité. En voici un exemple plus que classique :

f(x) ≜ if x=0 then 1 else x*f(x-1).

Cette définition de f utilise la définition de f ; en fait, la définition de f utilise une autre fonction, disons f', qui est définie comme f sauf en x et pour cela qu'on se permet de lui demander la valeur de f(x-1). La fonction f' va elle-même utiliser une fonction f'' un peu moins définie qu'elle et ainsi de suite jusqu'à la fonction nulle part définie. On a constitué une suite croissante de fonctions de mieux en mieux définie et la fonction f est l'"union" de toutes ces fonctions. Dans le terme "union" il y a une notion de limite et comme la mathématique nous a appris qu'une limite peut 'déborder', la notion de continuité va exprimer que la fonction définie préserve les limites : tout objet défini comme la limite d'une suite croissante dans un domaine a pour image dans la fonction la même valeur que la limite des images de la suite.

La fonction f exposée ci-dessus est continue ; voici une fonction qui ne l'est pas :

soit le domaine des nombres réels appartenant à [0..1] bornes incluses avec la relation d'ordre ≤ classique ; ce domaine est un espace inductif et l'on peut y définir des suites croissantes infinies comme par exemple la suite des x_i=0.5-1/n avec n nombre entier plus grand que 2.

soit la fonction g(x) ≜ if x<0.5 then 0 else 1.

Cette fonction est bien monotone, pas de problèmes de ce côté-là. L'image dans la fonction de tout objet de la suite est 0 et donc la limite de toutes ces images est 0. Par contre la limite de la suite croissante est 0.5 et donc l'image de cette limite 1. Mathématiquement, la fonction g est continue à droite mais pas à gauche.

Pourquoi interdire les fonctions non continues ?

La continuité n'a de sens que dans le cadre de domaines potentiellement infinis (sans quoi elle se confond avec celle de monotonie). Elle exprime que tout calcul de la fonction devra être effectué en un temps fini. Intuitivement, une fonction non continue en un point va demander une infinité d'approximations pour atteindre la valeur en ce point. Cette fonction n'est pas 'calculable' au sens qu'elle ne peut pas être définie comme la limite d'une suite de fonctions de plus en plus définies. Intuitivement, on peut dire aussi que si une fonction quelconque utilise une autre fonction par exemple dans : $f(x) \triangleq ... g(y) ...$ elle ne peut le faire que pour un nombre fini d'arguments et que donc, la fonction g_1 dont elle a besoin n'est qu'une fonction moins définie que g et que g_1 pourra être atteint par un nombre fini d'approximations successives : la limite d'une suite convergente.

L'écriture :

f : A→B est continue ⇔

$\forall\ x_1 \preceq x_2 \preceq x_3 \preceq x_4 \preceq ...$ avec $x_i \in A$ $\ \bigcup(f(x_i)) = f(\bigcup(x_i))$

doit être lue comme suit :

- on suppose toujours que f est au moins monotone ;
- si toute suite strictement croissante dans A est forcément finie, aucun problème ;
- intéressons-nous à un A tel qu'il puisse exister une suite strictement croissante infinie ; ce sera par exemple la cas si un des arguments de f est une fonction ; acceptons que l'argument est simplement h ; on a donc f(h). ;
- h n'est complètement connu que si l'on connaît toute sa table, or cette table peut être infinie ;
- f n'a pu consulter h que pour un nombre finis d'arguments, sans quoi f ne se termine pas ; appelons-les $h(k_1), h(k_2), ... h(k_n)$;
- f n'a donc eu besoin que d'une fonction h' plus petite que h : f(h) ne dépend que d'une partie finie de la table de h : la table de h' ;
- étant donné que la table de h' est finie, on ne peut construire qu'une suite finie de h_i et on a $h' = \bigcup h_i$ d'où $f(h) = f(h') = f(\bigcup h_i)$.

L'écriture suivante sera interdite :

f(x) := ... f(x+1) ...

puisque f demandera une consultation infinie de ... f

On peut démontrer qu'une composition continue de fonctions continues est continue ; par contre un programme informatique ne peut vérifier la continuité d'un programme aussi devra-t-on poser cette contrainte comme précondition .

calcul par point fixe

La théorie du POINT FIXE exposée en annexe, énonce que toute transformation continue défini un objet unique qui existe toujours : son plus petit point fixe et que cet objet peut être construit par approximations successives. Ne reprenons ici que ce que nous intéresse directement et est compréhensible intuitivement.

Une équation, quel qu'elle soit, peut être assimilée à une fonction ; une fonction qui à un argument associe un résultat. Si l'équation est du type "x:=1+2" il n'y a pas d'argument mais un résultat (fonction constante). Si l'équation est du type "f(x,y):=x+y" il y a deux arguments et un résultat. On peut simplifier, disant que toute équation du type "f(x,y):=..." est une suite finie ou infinie d'équations : une équation pour chaque combinaison possible d'arguments. Ainsi notre système se compose d'une suite finie ou infinie de fonctions constantes.

En fait, si chaque équation est continue comme exposé ci-dessus, la suite des fonctions constantes nécessaires sera finie.

La solution d'une équation appartient au domaine de l'équation, chaque domaine est muni d'une relation d'ordre partiel, et il y a toujours un plus petit élément noté $\perp$. On peut donc dire que la solution à chaque équation est "au moins" $\perp$.

Modifions légèrement notre définition de la relation d'ordre dans le domaine des fonctions A→B : f sera moins défini que g si pour tout x de a $f(x) \leq g(x)$ avec : si f n'est pas définie en x, on dira que $f(x)=\perp$.

La solution d'une équation est maintenant le plus grand élément d'une suite de fonctions (constantes) qui approximent cette solution.

Munissons chaque équation d'une <u>transformation</u>, c'est-à-dire d'une fonction dans le domaine des fonctions qui à une fonction moins définie 'retourne' une fonction mieux définie. Envisagé de cette manière, on construit une suite croissante de fonctions. La solution de l'équation sera la fonction telle que sa transformée ne donne pas de meilleure solution.

Que sera cette transformation ? tout simplement l'application de l'équation à la dernière valeur calculée ! Pas moyen de faire plus simple !

<u>algorithme</u>

Voici un exposé 'opérationnel' du programme :
. soit un système d'équations (que l'on peut même considéré comme infini)
. soit une question qui désigne une équation
. soit une table de réponses calculées initialisée à vide
. la réponse à la question sera calculée comme suit :
1) si l'équation n'a jamais été calculée on ajoute dans la table que sa réponse est $\perp$
2) on effectue le calcul de l'équation ; si ce calcul demande la valeur d'une autre équation, on effectue récursivement son calcul (calcul qui se terminera de par la propriété de continuité)
3) on compare le résultat obtenu avec celui déjà acquis dans la table ; si le résultat est meilleur on met à jour la table et on repasse à 2) sinon on met à jour la table et on sort (avec le résultat).

Cet algorithme se terminera si les équations fonctionnelles sont continues et s'il n'y a pas de formulation du type de "x:=[1|x]" qui provoquera un bouclage infini de 3) vers 2) puisqu'à chaque fois le résultat est "meilleur". Ceci dit, une formulation du type x:=[1|x] n'est pas continue puisqu'elle énonce une consultation infinie de la table ...

L'algorithme ne prétend pas rendre la meilleure solution, ainsi devant "x:=x" sera-t-il répondu $\perp$; il prétend seulement retourner <u>la plus</u>

<u>petite des solutions</u> (ce qu'on définira en annexe comme le plus petit point fixe).

On remarquera que cet algorithme donne une solution identique (pour la question posée) à celui qui effectue le calcul de chaque équation puis recommence jusqu'à ce qu'aucun résultat d'équation n'ait pu être amélioré. Il s'agit d'algorithmes "BOTTOM-UP" : qui partent de ⊥ pour remonter. Ce dernier algorithme n'est pas concevable en présence de variables de type fonction.

L'algorithme présenté n'est guère efficace : au regard d'un 'but' à atteindre, il y a des 'sous-buts' et la manière d'organiser ces sous-buts peut permettre d'éviter des recalculs inutiles. On discutera de ces algorithmes plus tard. Mais signalons déjà que l'algorithme choisi va dresser une carte de dépendances des variables d'équation, au fur et à mesure de la 'descente' d'une équation de manière telle à ce qu'un calcul d'équation ne soit réitéré que lorsque l'on soit sûr que toutes les données nécessaires ont atteint leur 'point fixe minimal'. L'algorithme est repris de <u>A Universal Top-Down Fixpoint Algorithm</u> de Baudouin Le Charlier et Pascal Van Hentenryck.

Des optimisations pourraient être introduites dans l'algorithme du fait qu'il a ici été décidé d'effectuer avant toute tentative d'exécution une analyse syntaxique de l'ensemble du système d'équations. L'analyse syntaxique est une 'interprétation' qui transforme le texte source de manière à le mettre sous une forme aisément exécutable : la notation polonaise postfixée. On pourrait profiter de cette 'avant-première' descente de chaque équation pour en analyser les propriétés. Nous présenteront quelques indications de recherche en ce sens puisqu'il s'agit de notre propos de départ : l'analyse statique.

<u>discussion quant'à ⊤ et ⊥</u>

Le concept de ⊥ a d'abord été introduit en informatique pour signifier qu'un programme boucle : un programme qui boucle retourne ⊥. Il n'y a pas moyen d'améliorer le résultat d'un programme qui boucle ! Notre ⊥ est à distinguer de ce 'bottom'-là sinon pour signifier qu'une fonction est mieux définie qu'une autre ; notre bottom est 'abstrait'.

A aucun moment ci-dessus la notion de ⊤ n'a été utilisée ; de fait ⊤ n'est pas nécessaire. Si nous l'avons introduit dans chaque domaine prédéfini du programme c'est par ce que les situations qui sont des impasses sont très courantes et que ⊤ permet de les dénoter. Ceci dit, la présence de ⊤ peut être totalement ignorée ; ce qui n'est pas le cas de ⊥ qui sera la valeur initiale de toute variable.

Que faire lors d'un if then else qui porte sur des expressions qui donnent ⊥ ou ⊤ ? Plusieurs interprétations monotones sont possibles. Nous avons décidé celle-ci :

```
    if a then c else d ≜
         if a=⊥ then ⊥
         else if a=⊤ then ⊤
              else if a=true then c  (quelque soit d)
                   else if a=false then d (quelque soit c)
```

Ce sera le cas quel que soit le type de c et d.
Cette définition du if imposera que tout symbole dans une équation soit typée ou puisse l'être avant exécution par l'analyseur de type (qu'on appelera analyseur sémantique), en effet en cas de rencontre d'un test qui donne ⊥ ou ⊤ il faut que le ifthenelse retourne le ⊥ ou ⊤ du bon type.

Le même problème se pose lors du test de comparaison :

```
a==b ≜
    if a=⊤ ∨ b=⊤ then ⊤ (le ⊤ du type booléen)
    else if a=⊥ ∨ b=⊥ then ⊥ (le ⊥ du type booléen)
        else a==b (qui retourne true ou false)
```

Cette interprétation est contraignante, on y considère qu'une comparaison portant sur ⊥ c'est comme une comparaison sur des données non initialisées.
L'écriture "x:=if a==⊥ then b else c", qui aurait permis de faire avancer la valeur de x, donnera toujours ⊥.
La solution sera d'utiliser l'opérateur "<<" qui désignera $\preceq$ dans le langage des équations : "if a<<⊥ then b else c" ou encore "if (a<<⊥) && (a>>⊥) then b else c" (confer plus loin les domaines Borne:, Gint: et Lint:).

Que faire lors de passages de paramètres ?
Le passage de paramètres se fera toujours par valeur, tous les paramètres seront évalués avant l'appel de la fonction et de gauche à droite. Un paramètre peut valoir ⊥ ou ⊤, l'appel sera effectué comme pour tout autre valeur.

Que faire lorsque ⊤ apparaît dans une expression ?
On pourrait considérer que si dans une expression apparaît la valeur ⊤ il n'y a plus moyen de l'améliorer et qu'il n'est donc plus nécessaire de poursuivre l'évaluation. En fait, cela n'est nullement exigé par la monotonie, pour preuve l'opérateur "glb" ou ">>" ou "plus grande borne inférieure" qui calcule le plus grand élément du domaine qui est plus petit que les deux : x>>⊤ donnera toujours x qui peut être $\prec$⊤.

Que faire lorsque ⊤ apparait dans un objet structuré ?
Il a été décidé que si ⊤ apparaît quelque part, n'importe où, dans un objet structuré tel que couple, n-uplet, liste, ... l'objet lui-même est à considérer comme ⊤. C'est-à-dire que lors de toute affectation à un objet de type structuré, le programme teste la présence de ⊤ et, le cas échéant, contraint toutes les parties à ⊤.

contrainte de monotonie

On a vu plus haut que le programme va construire une table des dernières approximations des variables d'équations ; or toute mise-à-jour de la table ne pourra se faire que de manière monotone : la nouvelle valeur doit être supérieure (ou égale) à l'ancienne. En ce qui concerne les variables-fonctions, si f(x) donne a, il faudra veiller à ce que pour tout f(y)=b existant dans la table et tel que $x \preceq y$ on ait bien $a \preceq b$; cette opération devra également être effectuée lors d'une première insertion : la valeur de départ de f(x) ne doit pas être $\perp$ mais a qui est la plus grande valeur des f(y) tel que $y \preceq x$.

Toutes ces opérations sont coûteuses en temps et il se peut qu'elles ne soient pas nécessaires. C'est inutile dans le cas de la fonction factorielle puisque deux arguments distintcs ne sont pas comparables (sauf envers $\perp$ et $\top$). C'est utile lorsque le domaine de la variable-fonction est un treillis à structure complexe : avec de nombreuses relations de dépendances entre objets, par exemple une fonction qui calcule une propriété d'objets qui sont des ensembles déjà en relation d'inclusion et telle que la propriété puisse être reportée.
La contrainte de monotonie va rendre inopérant l'usage de "variables temporaires' ; en effet, celles-ci ne pourront être correctement initialisées lors d'un nouvel usage si la valeur désirée est plus petite que celle attribuée lors de l'usage précédent. La contrainte de monotonie peut être coûteuse s'il est nécessaire de créer un domaine spécial, par exemple celui des compteurs en décrémentation ou pour un paramètre d'induction.

Pour toutes ces raisons, il sera autorisé dans le programme de déconnecter la contrainte de monotonie ; si les équations n'énoncent que des fonctions monotones, la contrainte de monotonie n'est pas nécessaire ; plus précisément, la loi de monotonie sous-entendue dans l'équation pourra être différente de celle appliquée par le programme.

la notion de treillis

Un treillis est un ensemble muni d'une relation d'ordre partiel tel qu'à tout couple d'éléments de l'ensemble corresponde un et un seul élément de l'ensemble qui soit le plus grand des plus petits des deux ainsi que un et un seul élément de l'ensemble qui soit le plus petit des plus grands des deux :
$(T,\preceq)$ ensemble (partiellement) ordonné est un treillis ssi

$$\forall a,b \in T\ \exists s \in T \mid a \preceq s \wedge b \preceq s \wedge (\forall s' \in T\ ((a \preceq s' \wedge b \preceq s)' \Rightarrow s \preceq s'))$$
$$\forall a,b \in T\ \exists i \in T \mid i \preceq a \wedge i \preceq b \wedge (\forall i' \in T\ ((i' \preceq a \wedge i' \preceq b) \Rightarrow i' \preceq i))$$

Il est complet s'il a un plus petit élément et un plus grand.

Le programme va exiger que tout domaine soit un treillis complet, en fait, il aurait suffit de la présence d'un plus petit élément et l'on peut ignorer la structure en treillis (pourvu qu'on crée un objet $\top$).

domaines d'application

Le logiciel, ici présenté, peut 'fonctionner' dans les domaines classiques tels que les entiers, les réels etc auxquels on a adjoint $\perp$ et $\top$ et qu'on appelle 'primitifs' ; toute variable sera alors définie à partir d'une fonction d'approximation qui part de $\perp$ et qui mène soit à $\perp$ (indétermination) soit à une valeur 'concrète' (comme dans le cas de la factorielle ci-dessus) soit à $\top$ (surdétermination).

Il peut être intéressant de 'voir' comment une fonction dont la table est infinie mais sera approximée en un temps fini pour le problème posé, de 'voir' comment le programme 'descend' les équations et compose son graphe de dépendances - son arbre de recherche - , de comparer par ce programme les graphes produits par différentes fonctions définissant la même table ; c'est pourquoi il est fourni une fonction de 'trace' qui permet de visualiser les calculs effectués et de recueillir quelques renseignements (rudimentaires) tels que le nombre de calcul d'équations, le nombre d'appels récursifs.

Mais nous sommes encore loin de notre propos de départ : l'analyse statique de programme par interprétation abstraite. C'est que ce travail ne fournit qu'un outil qui n'apporte en soi aucune solution immédiate. La tâche essentielle d'une analyse statique (étude d'un programme avant exécution) est la définition du quoi? : que cherche-t-on ? En cela ce programme n'est d'aucune utilité, il propose simplement un comment? Et ce comment est loin d'être immédiat : il faut en effet que :

- soient implémentés des domaines abstraits
  quelques domaines sont fournis
- soient implémentés des opérateurs sur ces domaines abstraits
  quelques opérateurs sur domaines prédéfinis sont fournis
- le problème puisse être exprimé sous forme
  d'un système d'équations dans ces domaines avec ces opérateurs

Le plus gros du travail est bien évidemment l'expression du problème sous forme d'équations. Dans le chapitre EXEMPLES nous proposerons une démarche générique, mais il est temps d'exposer le logiciel proprement dit.

PRESENTATION DU LOGICIEL

Le logiciel se compose des modules suivants :

- module intégrateur
  il est assez peu développé mais permet de spécifier un lexique de symboles et d'opérateurs et d'introduire un texte qui sera successivement fourni aux analyseurs lexical, syntaxique, sémantique puis permettra de demander la valeur d'une variable, tâche qui sera exécutée par le processeur de point fixe avec affichage de quelques renseignements élémentaires quant aux ressources utilisées ;

- analyseur lexical
  exécute les quelques commodités d'édition fournies et isole tous les identificateurs ; il ne demande que la mise à disposition d'un "lexique" ;

- analyseur syntaxique
  vérifie le respect de la syntaxe imposée du langage et construit un système d'équations dans le format interne des données en attribuant à chaque donnée le type explicite s'il est quelque part déclaré ou le type par défaut s'il a été demandé ; déclare les fonctions virtuelles si aucun type explicite n'a été détecté ; accepte les expressions parenthèsées et gère la précédence d'opérateurs ; le format interne des équations est en notation polonaise postfixée ;

- analyseur sémantique
  doit attribuer un type précis à chaque donnée du système d'équations ;

- module d'exécution
  le processeur de point fixe ; il calcule le plus petit point de fixe de toute fonction récursive dont l'"adresse" est transmise en paramètre ; la fonction reçoit à son tour l'adresse d'un programme qu'elle doit exécuter lors de tout appel récursif ; ce module est indépendant des autres, on notera en particulier que si le domaine des fonctions n'est pas autorisé dans un système d'équations autrement que comme type d'une variable d'équation (pas d'un paramètre formel), il l'est dans ce module : on peut avoir des fonctions de fonctions (ce n'est en fait qu'une table qui est manipulée).

- différents modules correspondant chacun à un domaine prédéfini et aux opérateurs qui l'accompagnent.

A l'heure de la remise de ce document, tous les codes du programme ont été rédigés excepté ceux de l'analyseur sémantique, si bien que le programme fonctionne en mode "virtuel" c'est-à-dire que :

- on doit définir le type de chaque variable de programme (mais n'importe où dans le texte et l'on peut faire usage d'un type par défaut)
- les opérateurs et fonctions prédéfinis du langage ne demandent aucun type explicite ; il y a branchement à la fonction correspondant aux types effectifs à l'exécution (comme le programme a été rédigé en C++, langage orienté objet, il est simplement fait usage de l'opérateur "virtual").

SYNTAXE DES EQUATIONS

Les symboles en **ombrés** sont 'terminaux' : se retrouveront dans le texte ; alors que les mots entre <> sont 'non-terminaux' : définis par ::=. Tout saut de ligne simple sera assimilé à un ∧ comme |.

```
<système>     ::=  <suite_def>.
                   <suite>.

<suite_def>   ::=  <rien>
                   <definition>
                   <definition><suite_def>

<definition>  ::=  #<nom>#<texte>#
                   ##<type>#
                   ##<nombre>,<nombre>#
                   #M0# | #M1# | #T0# | #T1#

<suite>       ::=  <suite_def><équation>
                   <suite_def><équation>;<suite>

<equation>    ::=  <var_tête>:=<suite_expr>

<var-tête>    ::=  <type><nom>
                   <typ_fct><nom>(<suite_form>)

<suite_form>  ::=  <type><nom>
                   <type><nom>,<suite_form>

<suite_exp>   ::=  <expression>
                   output(<expression>),<suite_exp>
                   <expression>,<suite_exp>

<type>        ::=  <rien>
                   Nul:
                   Top:
                   Pbool:
                   Pint:
                   Preal:
                   Pchar:
                   Pstring:
                   Gint:
                   Lint:(<entier>,<entier>):
                   Borne:(<type>):
                   Couple:(<type>,<type>):
                   Nuple:(<suite_type>):
                   Union:(<entier>,Nuple:(<suite_type>):):
                   Seq:(<type>):
                   Set:(<type>):

<type_fct>    ::=  <rien>
                   Fonct:(<type>-><type>):
                   Fonct:(CP:(<suite_typ>):-><type>):

<suite_type>  ::=  <type>
                   <type>,<suite_type>
```

```
<expression> ::=  <terme1><reste1>

<reste1>     ::=  <op1><terme1> | <rien>
<terme1>     ::=  <terme2><reste2>
......
<restei>     ::=  <opi><termei> | <rien>
<termei>     ::=  <termei+1><reste1+1>
......
<resten>     ::=  <opn><termen> | <rien>
<termen>     ::=  <base><resten>

<base>       ::=  (<expression>)
                  <fonction>
                  <variable>
                  <constante>
                  input
                  <type>if<expression>then<expression>
                                      else<expression>endif

<opi>        ::=  <nom>
                  <type_fct><nom>

<fonction>   ::=  <type_fct><nom>()
                  <type_fct><nom>(<suite_exp>)

<variable>   ::=  <type><nom>
                  <type_fct><nom>(<suite_parm>)

<var_queue>  ::=  <type><nom>

<suite_parm> ::=  <var_queue>
                  <var_queue>,<suite_param>

<nom>        ::=  <alpha><texte>

<alpha>      ::=  tout caractère
                  sauf séparateur, chiffre et symbole du langage

<texte>      ::=  <rien>
                  [<texte>]
                  <lettre><texte>

<lettre>     ::=  tout caractère sauf séparateur et symbole du langage

<constante>  ::=  <entier>
                  <type>[<texte>]

<entier>     ::=  <chiffre>
                  <chiffre><entier>

<chiffre>    ::=  0..9
```

<rien> signifie rien ! c'est-à-dire le vide.

COMMANDES PRE-LEXICALES

{ ... }

A tout endroit dans le texte, on peut insérer un commentaire entre {}. Un commentaire comporte toute chaîne sauf }. Le texte entre {} est tout simplement ignoré.

#<nom>#<texte>#

A partir du lieu d'insertion de **#**<nom>**#**<texte>**#**, toute occurrence de la partie gauche (<nom>) sera remplacée par la partie droite (<texte>). Une telle 'définition' n'est active que dans le texte qui suit et ne peut plus être modifiée. Les définitions peuvent être insérées n'importe où, mais il est bien entendu préférable de les regrouper en début de texte. Une portion de texte générée par substitution ne fera plus l'objet de substitution.

Exemple :

```
#I#Pint:#
#B#Pbool:#
#U#Nuple:(Pint:,Pint:,Pint:,Pbool:):#
U x1 := x2+x3 ;
U x2 := Nuple(x4,2,3,x5) ;
U x3 := x2+x2 ;
I x4 := 10 ;
#vrai#Pbool:[TRUE]#
B x5 := vrai.
```

La troisième ligne ne pouvait être #U#Nuple(I,I,I,B)#
Dommage !

##<type>#

##<type>**#** définit le type par défaut de toutes les variables, if_then-_else et constantes. Il ne peut y avoir qu'une seule occurrence de **##**<type>**#** dans tout le texte ; elle peut être insérée n'importe où ; mais de préférence au tout début.
En l'absence de toute commande **##**<type>**#**, il sera considéré que les fonctions peuvent être virtuelles, c'est-à-dire que la fonction à exécuter sera déterminée à l'exécution ; inversément, toute commande **##**<type>**#** interdira les fonctions virtuelles.

##<nombre>,<nombre>#

Détermine les valeurs $\perp$ et $\top$ des objets de type Gint:.
Confer plus loin ...

#M0# | #M1# | #T0# | #T1#

Interrupteur pour Monotonie et Trace d'exécution ; 0=non, 1=oui.

CORRECTION LEXICALE

L'analyse lexicale a pour but :
de définir la suite des symboles que sont

- les symboles du langage (en gras dans la syntaxe ci-dessus)
  ( ) , ; . [|] : if then else endif
  et les identificateurs de type : Pbool: Pint: etc
- les identificateurs d'opérateur ou de fonction
- les identificateurs de variable
- les constantes numériques
- les constantes entre [ ]

Les séparateurs de symbole sont tous ceux reconnus par la fonction standard du langage C int isspace(char) de <ctype.h> c'est-à-dire :
' ' '\t' '\f' '\n' '\r'
ainsi que les symboles en gras ci-dessus.

Les symboles du langage et les identificateurs de type sont énoncés dans un "lexique des symboles" ; ce lexique est redéfinissable de manière à
- modifier le texte de certains symboles
  := -> =   if/then/else/endif -> si/alors/sinon/finsi
- autoriser ou interdire certains types.

Il n'y a pas de contrainte particulière sur les identificateurs de symboles ; la chaîne la plus longue est recherchée en priorité : par exemple "=" et "==" peuvent coexister. Les symboles du langage sont identifiés en priorité ; aucun identificateur qui n'est pas un identificateur d'un symbole du langage ne peut comporter de séquence de caractères correspondant à un identicateur de symboles du langage.

Les symboles désignant une fonction ou un opérateur dans un certain domaine sont regroupés dans un "lexique des fonctions prédéfinies". A un même identificateur peuvent correspondre plusieurs fonctions ou opérateurs pourvu que ceux-ci se distinguent de par le type de leur domaine. Un identificateur de fonctions peut comporter une séquence de caractères correspondant à une autre fonction mais pas à un symbole du langage.
Les identificateurs de variables sont soumis aux contraintes lexicales suivantes :
- le premier caractère ne peut être un chiffre ;
- ils ne peuvent comporter aucune séquence de caractères désignant une fonction prédéfinie ou un symbole du langage ;
- la portée d'un identificateur de variable est tout le système d'équations ; aussi est-il interdit d'attribuer des identificateurs identiques à des variables désignant un paramètre formel ; ceci est interdit :

```
f(x,y) := ... ;
g(x) := ... .
```

il est conseillé de faire précéder l'identification d'un paramètre formel de l'identification de la fonction :

```
f(f_x,f_y) := ... ;
g(g_x) := ...
```

On peut donc indifféremment écrire

```
x1:=x2+2-azer348x(Bc,c,c+Bc)
x1 := x2 + 2 - azer348x ( Bc , c , c + Bc )
```

pour autant que + et - soient des opérateurs prédéfinis et que azer348x soit une fonction prédéfinie ou une variable-fonction définie.

CORRECTION SYNTAXIQUE

Confer la syntaxe exposée plus haut.

Syntaxiquement, il faut que
- chaque variable soit définie par une et une seule équation
- chaque fonction prédéfinie ou chaque variable de type fonction soit suivie d'autant de paramètres effectifs que dans sa définition
- chaque définition de type doit être correcte au regard de la définition récursive qui sera exposée un peu plus loin.
- une variable simple ne peut être de type Fct:
- une fonction ou une variable-fonction ne peut être déclarée d'un type autre que Fct:
- un même objet ne peut avoir deux définitions de type distinctes.

Toute définition de type d'un objet sera syntaxiquement reportée à tout endroit du texte où il est fait mention de cet objet. En cas d'absence de définition de type par défaut, c'est le type Nul: qui est attribué.

Nous avons vu que les fonctions et opérateurs autorisés sont regroupés dans un "lexique". Les "fonctions" sont préfixées ; les "opérateurs" sont soit binaires et infixés, soit unaires et préfixés. Tous les opérateurs sont qualifiés d'une "précédence". La précédence des opérateurs détermine l'ordre d'évaluation d'expressions non parenthésées : a+b*c*d-e deviendra (a+((b*c)*d))-e. Les opérations de même niveau de précédence sont exécutées de gauche à droite. Syntaxiquement donc, il y a insertion de parenthèses à tous les endroits du texte jusqu'à ce qu'il n'y ait plus - comme expression - que des unités syntaxiques indécomposables à savoir :
unité ::=
- [<type><constante>]
- [<type><variable>]
- [<type><variable>(<suite_d'unités>)]
- [<type><fonction_prédéfinie>(<suite_d'unités>)]
- [<unité><type><opérateur_binaire><unité>]
- [<type><opérateur_unaire><unité>]
- [<type>if<unité>else<unité>then<unité>endif]

Concrètement, chaque expression est transformée en notation polonaise postfixée et il est accepté qu'à une variable (simple ou fonction) soit 'affectée' une valeur qui est une suite d'expressions ; on verra que, sémantiquement, chaque expression sera évaluée (avec éventuellement des effets de bord) mais que seule la valeur calculée de la dernière expression sera assignée à la variable.

CORRECTION SEMANTIQUE

Un texte peut être syntaxiquement correct mais inexécutable ; celui-ci par exemple :

```
Pbool:a := b*3 ;
Pint:b := if b then [TRUE] else [un texte] endif.
```

Avant d'exposer les contraintes sémantiques exposons ce qu'on entendra ici par "fonction virtuelle".

Dans le lexique des fonctions prédéfinies nécessaire à l'analyse syntaxique peuvent apparaître plusieurs fonctions de même identificateur , ces fonctions seront toutefois classées par :

. leur arité (nombre de paramètres)
. le type du domaine

de telle sorte qu'il n'existe jamais deux fonctions de même identificateur, arité et type des arguments formels.
S'il existe deux fonctions de même identificateur et de même arité, on considérera qu'il existe une autre fonction de même identificateur et de même arité dont le rôle sera d'étudier à l'exécution le type des paramètres pour effectuer un branchement à la fonction dont le type correspond.

Lorsque l'analyse syntaxique rencontre un objet de type non spécifié, cet objet sera déclaré de type

Nul: pour variable simple, constante et paramètre formel
Fct:(type:->Nul:) pour variable-fonction et fonction
avec type: étant
Nul: si l'arité est de 1
CP:(Nul:,Nul:,...) si l'arité est >1.

Remarque : CP: est une variante de Nuple: qui ne sert qu'à distinguer les fonctions à un seul paramètre qui est un Nuple des fonctions à plusieurs paramètres insérés dans un Nuple.

Il faut que les contraintes de type suivantes soient observées :

- le type de l'expression qui suit **if** doit être Pbool: ;
- le type de l'expression qui suit **then** doit être identique au type de l'expression qui suit **else** ;
- le type de chaque sous-expression dans la suite parenthésée d'un argument effectif de variable-fonction ou de fonction doit être celui de la déclaration ;
- les types à gauche et à droite d'un **:=** d'une équation définissant une variable simple doivent être identique ; s'il est fait usage de l'opérateur **,** c'est le type de l'expression la plus à droite qui est utilisé ;
- le type du domaine d'une variable d'équation de type fonction doit correspondre au type de l'expression située à droite de **:=** dans sa définition ; s'il est fait usage de l'opérateur **,** c'est le type de l'expression la plus à droite qui est utilisé.

Pour rappel, les équations doivent définir des fonctions monotones et continues. Comme un système d'équations peut être une transformation continue alors que certaines fonctions ne sont pas monotones (au regard de la relation d'ordre implémentée !), on peut demander la déconnexion de la contrainte de monotonie ; cette déconnexion concernera cependant tout le système d'équations.

SIGNIFICATION DES TYPES, FONCTIONS ET OPERATEURS PREDEFINIS

La sémantique complète de l'"exécution d'un système d'équations" sera exposée plus loin ; nous n'exposerons ici que la sémantique d'une spécification de type ou d'un appel à une fonction.

L'évaluation d'une expression (a,b,c,d) se fait toujours <u>de gauche à droite et intégralement</u>.
Les "suites_d'expression" (expressions séparées par des virgules) ne sont autorisées qu'en paramètre d'une fonction ou variable-fonction ou comme partie droite d'une équation ; dans ce dernier cas, seule la valeur de la dernière expression sera retournée, les autres seront perdues (mais il y aura peut-être eu des effets de bord, confer output).
La transmission des paramètres à une variable-fonction ou à une fonction/opérateur prédéfini s'effectue toujours <u>par valeur</u>.
Il n'y a jamais d'abandon d'évaluation, comme on aurait pu le faire avec :   Pbool:a := a && [FALSE] && c && d ;

input

Cette fonction sans argument va recevoir une constante depuis le clavier. Sa signification est claire mais mérite un exemple :

a := input ; b := a*a ; c := b*b .

Le calcul de c ne va provoquer qu'une seule interrogation au clavier ; en effet la valeur assignée à a ne dépend d'aucune autre variable, aussi sa valeur sera-t-elle déclarée STABLE et toute requête ultérieure sera une consultation de table sans recalcul.

<type>

Tout opérateur, variable, fonction ou constante peut être précédé d'une spécification de type. Ce n'est pas obligatoire, l'analyse syntaxique puis sémantique tenteront de définir les types.

Un <type> dénote un ensemble de valeurs muni d'une relation d'ordre partielle $\preceq$, et d'un plus petit élément $\perp$ et tel que pour tout objet de type il ne puisse être construit de suite strictement décroissante infinie.

A chaque type correspond, dans le langage de description de système d'équations

- un identificateur de type   exemple : Pint: Pbool:
- un constructeur   exemple : Nuple(...)
- une série d'opérateurs et de fonctions prédéfinis.

Un constructeur est une fonction prédéfinie dans le "lexique", son nom est celui du type sans les ':' ; mais l'usage d'un constructeur explicite n'est pas indispensable si la valeur est un nombre. Concrètement, toute séquence de caractères débutant par un chiffre est transformée en valeur du type Pint: et toute séquence entre [] est transformée en valeur du type de la variable ; c'est le symbole | qui sert à structurer une constante entre []. Entre les [] ne peuvent être utilisés de variables.

Les opérateurs et fonctions prédéfinis seront énoncés lors de l'exposé de chaque type. Sauf l'opérateur implicite d'assignation de valeur := ; cet opérateur est en fait très spécial puisqu'il effectue

- un lub sur la dernière estimation de telle sorte que la monotonie soit toujours assurée. Ce calcul de lub n'est pas effectué au sein des routines du domaine en question mais par la routine 'tau' de l'algorithme de point fixe.
- un test de valeur ⊤.

(tout ceci n'est bien entendu pas effectué lors de l'affectation de paramètres formels)

Pour toutes les définitions qui suivent, les mots du langage de description d'un système d'équations débutent par une majuscule. Par contre les mots du métalangage d'exposé de la sémantique débutent par une minuscule.

On suppose déjà définis
    les ensembles de valeurs :
        boolean = { true,false }
        integer real string char
    et les fonctions traditionnelles sur ces valeurs que sont
        + - * / % && || ^^ !
        ainsi que la concaténation de deux string
        et l'extraction de la première occurrence d'un string
            dans un string
    un type Texte
    les opérateurs de métalangage
        $\in$ $\forall$ $\exists$ $\wedge$ $\vee$ $\veebar$ $\bigcup$ $\bigcap$ $\uplus$ $=$ $\neq$ < > $\leq$
        if then else

Ainsi dans ce métalangage pourra-t-on écrire par exemple :
    $\forall$ x $\in$ integer ou $\forall$ x $\in$ Pint:

On remarquera que les fonctions sont spécifiées de manière mathématique : il n'y a pas de "préconditions"/"postconditions".
Exemple : la fonction Subst_ieme qui permet de substituer un caractère au milieu d'un 'string' est définie par :

```
res := Fonct:(CP:(Pint:,Pstring:,Pchar:):->Pstring:): Subst_ieme(n,s,c)
     if n=⊤ ∨ s=⊤ ∨ c=⊤ ∨ n≤0 ∨ n>len(s) then res=⊤
     else if n=⊥ ∨ s=⊥ ∨ c=⊥ then res=⊥
          else res=s[1..n-1]+c+s[n+1..len(s)]
```

Les fonctions sont totales dans la mesure où l'on accepte que tous les types ont été correctement définis (en fait, le programme ne contrôle pas - pour l'instant - que lors de la fonction Test-Ieme sur Nuple tous les sous-types sont identiques).

On acceptera dans l'exposé qui suit qu'un opérateur binaire @ puisse être utilisé sous sa forme infixe a@b comme préfixe @(a,b).

Les fonctions sémantiques de métalangage len() et de concaténation sur chaînes sont supposées définies et il n'est nul besoin de "préconditions".

Bien plus d'opérateurs sont définis dans les domaines qu'énoncés ci-dessous ; n'ont en fait été repris que les opérateurs accessibles depuis un système d'équations : dans le langage du système d'équations. Les autres opérateurs ne sont accessibles que dans le langage du compilateur : le langage C++ ; ils servent lorsque l'on s'adresse directement au processeur de point fixe ou lorsque l'on désire implémenter un nouveau

type, un nouvel opérateur ou fonction ; on trouvera ces opérateurs définis dans le listing du programme au niveau de la définition de "classe" du domaine.

Les types suivants sont des domaines qui existent bel et bien, mais ne sont pas accessibles dans un système d'équations ; ils ne le sont qu'en programmation dans le langage du compilateur ; ce qu'on n'abordera pas ici, ce serait trop long :

```
Nul: , Top:     (servent lors des analyses de types)
Fct:            (type fonction sans table)
Fonct:          (type fonction avec table ; sous-type de Fct:)
Algo0:, Algo1:,Algo2:
                (sous-types de Fonct:, à qui est associé un algo-
                rithme de calcul de plus petit point fixe et toute
                la batterie annexe : table des dépendances etc.)
```

Signalons par exemple qu'un système d'équations est un objet du type Algo2:, sous-type de

Fct:(Couple:(Pint:,Union:(D_1,D_2,..D_n):):->Union:(C_1,C_2,..C_n):):)

avec n le nombre de variables d'équation, Pint: le type d'un objet où est rangé le numéro de la variable, D_i le type du domaine de la ième variable et C_i le type de son codomaine.

Les opérateurs prédéfinis sont rangés en ordre de précédence (un nombre entier), les voici :

```
!    préfixe  10  (le not booléen)
--   préfixe  10
++   préfixe  10
*    infixe   20
/    infixe   20
%    infixe   20
+    infixe   30
-    infixe   30
<<=  infixe   40  (calcule le lub : plus petite borne supérieure)
>>=  infixe   40  (calcule le glb : plus grande borne inférieure)
<    infixe   50
>    infixe   50
<=   infixe   50
>=   infixe   50
<<   infixe   50  (relation d'ordre partielle ≤)
>>   infixe   50  (la même dans l'autre sens, pas très utile ...)
==   infixe   60  ( a==b n'a pas le même sens que a<<b && a>>b )
!=   infixe   60
&&   infixe   80  ( le 'et' : ∧ )
||   infixe   80  ( le 'ou' : ∨ )
^^   infixe   80  ( le 'ou' exclusif : ⊻ , également sur Set:)
```

On trouvera cette liste accompagnée des fonctions dans le "Lexique standard des fonctions" c'est-à-dire la table des fonctions autorisées.

Seuls les opérateurs <<= et >>= sont totaux (partout définis) ; les autres exigent que les types correspondent. Si a et b sont de types distincts a<<=b créera un objet $\top$, a>>=b un objet $\bot$, il s'agit d'objets d'un 'autre type', les types Nul: et Top: qui ne sont pas normalement accessibles dans un système d'équations ; ce genre de situation ne peut se produire que dans un cas : la comparaison d'objets d'un même type Union mais de sous-type distincts.

Les fonctions prédéfinies sont :
Pbool, Pint, Pchar, Pstring, Gint, Lint, Couple, Borne, Nuple, Union, Taille, Ieme, Subst_ieme, Subst_mot, Test_mot, Gauche, Droite, Nbu.

Les domaines primitifs

Il y a 5 types de base, ce sont les "domaines primitifs" :

Pbool: Pint: Preal: Pchar: Pstring:

Ces domaines primitifs désignent toutes les valeurs traditionnelles des domaines boolean integer real string char complétés de $\perp$ et $\top$.

Pour rappel :

$\text{Pint:} = \text{integer} \cup \{\perp,\top\}$

$\forall\ x,y \in \text{integer}\ x \not\preceq y \wedge x \not\succeq y$

$\forall\ x \in \text{Pint: } \perp \preceq x \wedge x \preceq \top \wedge x \preceq x$

$x \not\preceq y \wedge x \not\succeq y$ signifie que la relation d'ordre $\preceq$ n'est pas établie entre x et y.

On crée un objet de type Pint: (par exemple) par :

Pint:x := 1.	un nombre devient d'office Pint:
Pint:x := [1].	transfert de type <texte> --> Pint:
Pint(1).	appel explicite du constructeur

On remarquera plus loin que les opérateurs % et / donnent :

a/0 = TOP
a%0 = TOP.

Dans les domaines primitifs, toute fonction monotone est continue (toute chaîne est stationnaire) ; on veillera donc simplement à vérifier la monotonie de chaque opérateur.

Les quatre opérateurs suivants ont même sémantique quelque soit le domaine primitif :

Opérateur infixe <<= :

Cet opérateur retourne le lub de deux valeurs de même domaine primitif ; exemple pour Pint: :

```
res := Fonct(CP:(Pint:,Pint:)->Pint:): a<<=b
     if a=⊤ ∨ b=⊤ then ⊤
     else if a=⊥ then b else if b=⊥ then a else if a=b then a else ⊤
```

Opérateur infixe >>= :

Cet opérateur retourne le glb de deux valeurs de même domaine primitif ; exemple pour Pint: :

```
res := Fonct(CP:(Pint:,Pint:)->Pint:): a>>=b
     if a=b then a else if a=⊥ ∨ b=⊥ then ⊥ else if a=⊤ then b else a
```

Opérateur infixe << :

```
res := Fonct(CP:(Pint:,Pint:)->Pint:): a<<b
     if a=b then true else if a=⊥ then true else b=⊤
```

Opérateur infixe >> :

```
res := Fonct(CP:(Pint:,Pint:)->Pint:): a>>b
     if a=b then true else if a=⊤ then true else b=⊥
```

Le type Pbool

Constructeur Pbool :

```
res := Fonct:(Texte->Pbool:): Pbool(@)
     avec @ désignant [BOTTOM] ou [TOP] ou [TRUE] ou [FALSE]
```

Opérateur préfixé ! :

```
res := Fonct:(Pbool:->Pbool:): !(a)
     if a=⊥ then res=⊥ else if a=⊤ then res=⊤ else res=!a
```

Opérateurs infixés && || ^^ :

```
Ci-dessous, on remplacera @ par un des symboles bien connus
     && || ^^  en considérant &&=∧ ||=∨ ^^=⊻
res := Fonct:(CP:(Pbool:,Pbool:):->Pbool:): @(a,b)
     if a=⊤ ∨ b=⊤ then res=⊤ else if a=⊥ ∨ b=⊥ then res=⊥ else res=a@b
```

Opérateurs infixés == != <= >= < > :

```
Ci-dessous, on remplacera @ par un des symboles bien connus
     == != <= >= < >      (!= devient ≠)
res := Fonct:(CP:(Pbool:,Pbool:):->Pbool:): @(a,b)
     if a=⊤ ∨ b=⊤ then res=⊤ else if a=⊥ ∨ b=⊥ then res=⊥ else res=a@b
```

```
Remarque : les opérateurs ci-dessus (&& || ^^ == != <= >= < >) désignés
par @ sont bien monotones :
     exemples :      (⊥@⊥)≼(⊥@x)≼(x@x)≼(⊤@x)≼(⊤@⊤)
                       ⊥  ≼  ⊥  ≼  y  ≼  ⊤  ≼  ⊤
     remarquons qu'il a été décidé que (⊥@⊤) donne ⊤
```

Le type Pint:

Constructeur Pint :

```
     Il y a un constructeur implicite :
          celui lancé lexicalement à la rencontre d'un chiffre
          mais le nombre ne pourra être que positif.
     Sinon, le constructeur sera simplement :
res := Fonct:(Texte->Pint:): Pint(@)
          [BOTTOM] ou [TOP] ou un nombre encadré par []
```

Opérateurs préfixés ! ++ -- :

```
Ci-dessous, on remplacera @ par l'un des symboles bien connus :
     ! ++ --
res := Fonct:(Pint:->Pint:): @(a)
     if a=⊥ then res=⊥ else if a=⊤ then res=⊤ else res=@a
```

Opérateurs infixés && || ^^ + - * :

```
Ci-dessous, on remplacera @ par l'un des symboles bien connus :
     && || ^^ + - *
res := Fonct:(CP:(Pint:,Pint:):->Pint:): @(a,b)
     if a=⊤ ∨ b=⊤ then res=⊤ else if a=⊥ ∨ b=⊥ then res=⊥ else res=a@b
```

Opérateurs infixés / % :

Ci-dessous, on replacera @ par l'un des symboles bien connus :

```
      / %
res := Fonct:(CP:(Pint:,Pint:):->Pint:): @(a,b)
      if a=⊤ ∨ b=⊤ ∨ b=0 then res=⊤ else if a=⊥ ∨ b=⊥ then res=⊥
                                                       else res=a@b
```

Opérateurs infixés == != <= >= < > :

Ci-dessous, on remplacera @ par un des symboles bien connus

```
      == != <= >= < >
res := Fonct:(CP:(Pint:,Pint:):->Pbool:): @(a,b)
      if a=⊤ ∨ b=⊤ then res=⊤ else if a=⊥ ∨ b=⊥ then res=⊥ else res=a@b
```

Le type Preal:

Constructeur Preal :

```
res := Fonct:(Texte->Preal:): Preal(@)
      avec @ : [BOTTOM] ou [TOP] ou [<nombre>]
```

Opérateurs préfixés ++ -- :

Ci-dessous, on remplacera @ par l'un des symboles bien connus :

```
      ++ --
res := Fonct:Preal:->Preal:): @(a)
      if a=⊥ then res=⊥ else if a=⊤ then res=⊤ else res=@a
```

Opérateurs infixés + - * :

Ci-dessous, on remplacera @ par l'un des symboles bien connus :

```
      + - *
res := Fonct:(CP:(Preal:,Preal:):->Preal:): @(a,b)
      if a=⊤ ∨ b=⊤ then res=⊤ else if a=⊥ ∨ b=⊥ then res=⊥ else res=a@b
```

Opérateur infixé / :

```
res := (Fonct:CP:(Preal:,Preal:):->Preal:): @(a,b)
      if a=⊤ ∨ b=⊤ ∨ b=0 then res=⊤ else if a=⊥ ∨ b=⊥ then res=⊥
                                                       else res=a/b
```

Opérateurs infixés == != <= >= < > :

Ci-dessous, on remplacera @ par un des symboles bien connus

```
      == != <= >= < >
res := Fonct:CP:(Preal:,Preal:):->Pbool:): @(a,b)
      if a=⊤ ∨ b=⊤ then res=⊤ else if a=⊥ ∨ b=⊥ then res=⊥ else res=a@b
```

Le type Pchar:

Confer Pint:

Le type Pstring:

Constructeur :

```
res := Fonct:(Texte->Pstring:): Pstring(@)
      par convention [BOTTOM] et [TOP] sont réservés à ⊥ et ⊤
```

Opérateur infixé +

```
res := Fonct:(CP:(Pstring:,Pstring:):->Pstring:): +(a,b)
     if a=⊤ ∨ b=⊤ then res=⊤
     if a=⊥ ∨ b=⊥ then res=⊥ else res=concaténation_de(a,b)
```

Opérateurs infixés == != <= >= < > :

Ci-dessous, on remplacera @ par un des symboles bien connus

```
     == != <= >= < >
res := Fonct:(CP:(Pstring:,Pstring:):->Pbool:): @(a,b)
     if a=⊤ ∨ b=⊤ then res=⊤ else if a=⊥ ∨ b=⊥ then res=⊥ else res=a@b
```

Fonction Taille

```
res := Fonct:(Pstring:->Pint:): Taille(s)
     if s=⊥ then res=⊥ else if s=⊤ then res=⊤ else res=len(s)
On peut définir len(s) par len(s)=n tel que n[]=μL0s
     avec Ln[]=n[] ∧ Ln[c|s]=(n+1)s
```

Fonction Ieme

```
res := Fonct:(CP:(Pint:,Pstring:):->Pchar:): Ieme(n,s)
     if n=⊤ ∨ s=⊤ ∨ n≤0 ∨ n>len(s) then res=⊤
     else if n=⊥ ∨ s=⊥ then res=⊥ else res=s[n]
```

Fonction Subst ieme

```
res := Fonct:(CP:(Pint:,Pstring:,Pchar):->Pstring:): Subst_ieme(n,s,c)
     if n=⊤ ∨ s=⊤ ∨ c=⊤ ∨ n≤0 ∨ n>len(s) then res=⊤
     else if n=⊥ ∨ s=⊥ ∨ c=⊥ then res=⊥
          else res=s[1..n-1]+c+s[n+1..len(s)]
```

Fonction Subst mot

```
res := Fonct:(CP:(Pint:,Pint,Pstring:,Pstring):->Pstring:):
          Subst_mot(n,m,s,c)
     if n=⊤ ∨ m=⊤ ∨ s=⊤ ∨ c=⊤ ∨ n≤0 ∨ m<n ∨ m>len(s) then res=⊤
     else if n=⊥ ∨ m=⊥ ∨ s=⊥ ∨ c=⊥ then res=⊥
          else res=s[1..n-1]+c+s[m+1..len(s)]
```

Fonction Test mot

```
res := Fonct:(CP:(Pstring:,Pstring:):->Pint:): Test_mot(s,c)
     if s=⊤ ∨ c=⊤ then res=⊤
     else if s=⊥ ∨ c=⊥ then res=⊥ else res=occurrence_de(s,c)
```

occurence_de se retourne 0 si la séquence c n'existe nulle part en s, sinon l'indice de l'emplacement en s de la séquence c.

```
     occurrence_de(s,c)=n
     ⇔ ls=len(s) ∧ lc=len(c) ∧
          ∀ 0<i≤n ∃ o<j≤lc (s[i+j]≠c[j] ∨ i+j>ls) ∧
          ∀ 0<j≤lc s[n+j]=c[j]
```

Le type Gint:

Le type Gint: dénote tous les nombres entiers compris entre deux bornes définies globalement pour tout le système d'équations. La relation d'ordre est la relation traditionnelle ≤. ⊥ est la borne inférieure et ⊤ est la borne supérieure.

On crée un objet de ce type par
Gint(1..10):x := 20.   (ce qui donnera 10 !)

Les fonctions et opérateurs sont semblables à ceux du type Pint:, sauf qu'après chaque calcul il est effectué :
si le résultat est > au maximum alors maximum
si le résultat est < au minimum alors minimum.
On remarquera que pour effectuer un test sur Gint, il faut utiliser << et non ==.

<u>Constructeur Gint :</u>

res := Fonct:(Texte->Gint:): Gint(@)
avec @ : [BOTTOM] ou [TOP] ou un nombre encadré par []

<u>Opérateur préfixé ++ -- :</u>

res := Fonct:(Gint:->Gint:): ++(a)
r=a+1 ∧ res=if r<min then min else if r>max then max else r

<u>Opérateurs infixés + * -</u>

Ci-dessous, on remplacera @ par l'un des symboles bien connus + * -
res := Fonct:(CP:(Gint:,Gint:):->Gint:): @(a,b)
r=a@b ∧ res=if r<min then min else if r>max then max else r

<u>Opérateurs infixés == != <= >= < > :</u>

Ci-dessous, on remplacera @ par un des symboles bien connus
== != <= >= < >
res := Fonct:(CP:(Gstring:,Gstring:):->Pbool:): @(a,b)
if a=⊤ ∨ b=⊤ then res=⊤ else if a=⊥ ∨ b=⊥ then res=⊥ else res=a@b
On utilisera plutôt << et >>.

<u>Opérateurs infixés << >> :</u>

res := Fonct:(CP:(Gint:,Gint:):->Pbool:): <<(a,b)
res=a≼b
res := Fonct:(CP:(Gint:,Gint:):->Pbool:): >>(a,b)
res=b≼a

<u>Opérateurs infixés <<= >>= :</u>

res := Fonct:(CP:(Gint:,Gint:):->Pbool:): <<=(a,b)
res=lub(a,b)= if a≤b then a else b
res := Fonct:(CP:(Gint:,Gint:):->Pbool:): >>=(a,b)
res=glb(a,b)= if b≤a then b else a

<u>Le type Lint:</u>

Le type Lint: dénote tous les nombres entiers compris entre deux bornes définies localement. Il s'agit en fait d'une infinité de types ; ne sont comparables que les objets qui ont même borne inférieure et même borne supérieure. La relation d'ordre est la relation traditionnelle ≤. ⊥ est la borne inférieure et ⊤ est la borne supérieure.

Confer Gint:

Le type Couple:

Le type Couple: dénote tous les couples de deux objets.
Deux objets de type Couple: ne sont comparables que si leurs objets "gauche" et droite" sont respectivement comparables.

Désignons le premier élément du couple par gauche tel que si x est un couple alors gauche(x) est son premier élément
Désignons le second élément du couple par droite tel que si x est un couple alors droite(x) est son second élément.
Désignons enfin le couple formé de x et y par couple(x,y)
Ce sont, bien entendu, des fonctions de métalangage.
Ainsi

```
     c=couple(x,y) <=> x=gauche(c) ∧ y=droite(c)
```

Un objet de type Couple: est ⊥ si ses deux éléments sont ⊥.
est ⊤ si l'un de ses éléments est ⊤

```
          couple(x,y) ≤ couple(a,b)  ⇔   x≤a ∧ y≤b
          couple(x,y) ⇒  (x=⊤ ⇔  y=⊤)
```

⊥ désignera généralement la 'non-initialisation' ou 'on en connait rien' ; alors que ⊤ signale que ce couple n'est plus nulle part utilisable : il sèmera la 'zizanie' au sens de 'tout qui touche à ⊤ devient ⊤.

Fonction constructrice Couple :

```
res := Fonct:(CP:(Typeg:,Typed:):->Couple:(Typeg:,Typed:):):
                    Couple(a,b)
     if a=⊤ ∨ b=⊤ then res=⊤ else res=couple(a,b)
```

Fonction Droite :

```
res := Fonct:(Couple:(Typeg:,Typed):->Typed:): Droite(x)
     res=droite     (rappel : droite=⊤ ⇔  gauche=⊤)
```

Fonction Gauche :

```
res := Fonct:(Couple:(Typeg:,Typed):->Typeg:): Gauche(x)
     res=gauche     (rappel : droite=⊤ ⇔  gauche=⊤)
```

Opérateurs préfixés ! ++ -- :

Remplacez ci-dessous @ par l'un des ! ++ -- :

```
res := Fonct:(Couple:(Typeg:,Typed:):->Couple:(Typeg:,Typed:):): @(x)
     r=couple(@gauche(x),@droite(x)) ∧
     if (droite=⊤ ∨ gauche=⊤) then res=⊤ else res=r
```

remarque : si @ n'est pas un opérateur défini pour droite et gauche, il y a aura message d'erreur "opération non définie dans ce domaine" ; l'analyse sémantique a dû écarter ce genre de possibilité ; sinon, on peut toujours implémenter un opérateur @ qui retourne toujours ⊤.

Opérateurs infixés && || ^^ + - * / % :

Remplacez ci-dessous @ par l'un des && || ^^ + - * / % :

```
res := Fonct:(CP:(Couple:(Typeg:,Typed:):, Couple:(Typeg:,Typed:):):
               ->Couple:(Typeg:,Typed:):): @(x,y)
     r=couple(gauche(x)@gauche(y),droite(x)@droite(y)) ∧
     if (droite=⊤ ∨ gauche=⊤) then res=⊤ else res=r
```

Remarque : si l'on veut modifier un Couple, le plus simple est d'utiliser le constructeur comme suit :

```
     Couple:(Pint:,Pbool:): a := Couple(gauche(a),[FALSE]) ;
```

Opérateurs infixés == < > <= >= :

Ci-dessous, on remplacera @ par un des symboles == < > <= >=

```
res := Fonct:(CP:(Couple:(...):,Couple:(...):):->Pbool:): @(a,b)
     if a=⊤ ∨ b=⊤ then res=⊤
     else if droite(a)=⊥ ∨ droite(b)=⊥ ∨ gauche(a)=⊥ ∨ gauche(b)=⊥
          then res=⊥
          else res= (droite(a)@droite(b) ∧ gauche(a)@gauche(b))
```

Opérateurs infixés != :

```
res := Fonct:(CP:(Couple:(...):,Couple:(...):):->Pbool:): !=(a,b)
     if a=⊤ ∨ b=⊤ then res=⊤
     else if droite(a)=⊥ ∨ droite(b)=⊥ ∨ gauche(a)=⊥ ∨ gauche(b)=⊥
          then res ⊥
          else res= (droite(a)!=droite(b) ∨ gauche(a)!=gauche(b))
```

Opérateurs infixés << >>

```
res := Fonct:(CP:(Couple:(...):,Couple:(...):):->Pbool:): <<(a,b)
     res=(gauche(a)≤gauche(b))∧(droite(a)≤droite(b))
res := Fonct:(CP:(Couple:(...):,Couple:(...):):->Pbool:): <<(a,b)
     res=(gauche(b)≤gauche(a))∧(droite(b)≤droite(a))
```

Opérateurs infixés <<= >>=

```
res := Fonct:(CP:(Couple:(Typeg:,Typed:):, Couple:(Typeg:,Typed:):):
               ->Couple:(Typeg:,Typed:):): <<=(x,y)
     res=couple(lub(gauche(a),gauche(b)),lub(droite(a),droite(b))
res := Fonct:(CP:(Couple:(Typeg:,Typed:):, Couple:(Typeg:,Typed:):):
               ->Couple:(Typeg:,Typed:):): >>=(x,y)
     res=couple(glb(gauche(a),gauche(b)),glb(droite(a),droite(b))
```

Le type Borne:

Le type Borne: dénote tous les couples de deux objets de même type. Deux objets de type Borne: ne sont comparables que si leurs objets sont comparables. Un objet de type Borne:(x,y) dénote l'ensemble de tous les objets z tels que x≤z ∧ z≤y ; il n'est pas considéré 'objet structuré'. On a x≻y ⇔ Borne(x,y)=⊥. ⊤ est (⊥,⊤).

Fonction constructrice Borne :

```
res := Fonct:(CP:(Typeg:,Typed:):->Borne:(Typeg:,Typed:):): Borne(a,b)
     res=couple(a,b)
```

Fonction Droite :

```
res := Fonct:(Borne:(Typeg:,Typed):->Typed:): Droite(x)
     res=droite
```

Fonction Gauche :

```
res := Fonct:(Borne:(Typeg:,Typed):->Typeg:): Gauche(x)
     res=gauche
```

Opérateurs préfixés ++ -- :

Remplacez ci-dessous @ par l'un des ! ++ -- :

```
res := Fonct:(Borne:(Typeg:,Typed:):->Borne:(Typeg:,Typed:):):   @(x)
     r=couple(@gauche(x),@droite(x))
```

Opérateurs infixés + - * :

Remplacez ci-dessous @ par l'un des + - * :
```
res := Fonct:(CP:(Borne:(Typeg:,Typed:):, Borne:(Typeg:,Typed:):):
               ->Borne:(Typeg:,Typed:):): @(x,y)
     r=couple(gauche(x)@gauche(y),droite(x)@droite(y))
```

Opérateurs infixés == :

```
res := Fonct:(CP:(Borne:(...):,Borne:(...):):->Pbool:): ==(a,b)
     if a=⊤ ∨ b=⊤ then res=⊤
     else if a=⊥ ∨ b=⊥ then res ⊥
          else res= (droite(a)==droite(b) ∨ gauche(a)==gauche(b))
```
Y préférér l'usage de << et >>

Opérateur infixé !=

```
res := Fonct:(CP:(Borne:(...):,Borne:(...):):->Pbool:): ==(a,b)
     if a=⊤ ∨ b=⊤ then res=⊤
     else if a=⊥ ∨ b=⊥ then res ⊥
          else res= (droite(a)!=droite(b) ∨ gauche(a)!=gauche(b))
```
Y préférér l'usage de << et >>

Opérateurs infixés << >>

```
res := Fonct:(CP:(Borne:(...):,Borne:(...):):->Pbool:): <<(a,b)
     res=(gauche(b)≤gauche(a))∧(droite(a)≤droite(b))
res := Fonct:(CP:(Borne:(...):,Borne:(...):):->Pbool:): >>(a,b)
     res=(gauche(a)≤gauche(b))∧(droite(b)≤droite(a))
```

Opérateurs infixés <<= >>=

```
res := Fonct:(CP:(Borne:(Typeg:,Typed:):, Borne:(Typeg:,Typed:):):
               ->Borne:(Typeg:,Typed:):): <<=(x,y)
     res=couple(glb(gauche(a),gauche(b)),lub(droite(a),droite(b))
res := Fonct:(CP:(Borne:(Typeg:,Typed:):, Borne:(Typeg:,Typed:):):
               ->Borne:(Typeg:,Typed:):): >>=(x,y)
     res=couple(lub(gauche(a),gauche(b)),glb(droite(a),droite(b))
```

Le type Nuple:

Le type Nuple: dénote tous les nuples de n objets, avec n>0.
Il s'agit d'une infinité de types bien distincts, un objet de type Nuple: n'est comparable qu'avec un objet qui dispose du même nombre d'objets et dont les types des objets sont respectivement identiques.

```
x est un Nuple <=>  nbu(x) est son nombre d'éléments
                    ième(i,x) est son ième élément avec 0<i≤nbu(x)
```

La relation d'ordre partielle est définie par :
```
     x ≤ y <=> y=⊤ ∨
               ( nbu(x)=nbu(y) ∧
               ∀ i | 0<i<=nbu(x) type(ième(i,x))=type(ième(i,y) ∧
                  ième(x) ≤ ième(y) )
     ⊤ est défini par x=⊤ ⇔ ∃ 0<i<=nbu(x) ieme(i,x)=⊤
     et donc x=⊤ ⇔ ∀ 0<i<=nbu(x) ieme(i,x)=⊤
```

Fonction constructrice Nuple :

```
res := Fonct:(CP:(...):->Nuple:(...)
          suffit d'énumérer les données séparées par des virgules.
```

Fonction Nbu :

```
res := Fonct:(Nuple:(...):->Pint(1..n):): Nbu(x).
     if x=⊤ then res=⊤ else res=nbu(x)
```

Fonction Ieme :

```
res := Fonct:(Nuple:(...):->Typei:): Ieme(i,x).
     if x=⊤ ∨ i<0 ∨ i>nbu(x) then res=⊥Typei else res=ième(i,x)
```

Opérateurs préfixés ! ++ -- :

```
Ci-dessous, on remplacera @ par l'un des symboles bien connus :
     ! ++ --
res := Fonct:(Nuple:(...):->Nuple:(...)): @(a)
     if a=⊤ ∨ ∃ 0<i≤nbu(a) @(Ieme(i,a))=⊤ then res=⊤
     else res∈Nuple:(...): | ∀ 0<i≤<nbu(a) ieme(res)=@(Ieme(i,a))
rem : confer vérification sémantique des types
```

Opérateurs infixés && || ^^ + - * / % :

```
Ci-dessous, on remplacera @ par l'un des symboles bien connus :
     && || ^^ + - * / %
res := Fonct:(CP:(Nuple:(...):,Nuple:(...):):->Nuple:(...)): @(a,b)
     if a=⊤ ∨ b=⊤ ∨ ∃ 0<i≤nbu(a) @(Ieme(i,a),Ieme(i,b))=⊤
     then res= ⊤ else res∈Nuple:(...): |
                              ∀ 0<i≤<nbu(a) ieme(res)=@(ieme(i,a),ieme(i,b))
```

Opérateurs infixés <<= >>=

```
     remplacer @ par <<= ou >>=
res := Fonct:(CP:(Nuple:(...):,Nuple:(...):):->Nuple:(...)): @(a,b)
     ∀ 0<i≤nbu(a) res[i]=a[i]@b[i]
```

Opérateurs infixés << >>

```
     remplacer @ par << ou >>
res := Fonct:(CP:(Nuple:(...):,Nuple:(...):):->Nuple:(...)): @(a,b)
     res = ∀ 0<i≤nbu(a) a[i]@b[i]
```

Opérateurs infixés == < > :

Ci-dessous, on remplacera @ par un des symboles bien connus
== < >

```
res := Fonct:(CP:(Nuple:(...):,Nuple:(...):):->Pbool:): @(a,b)
     if a=⊤ ∨ b=⊤ then res=⊤
     else if ∨ ∃ 0<i≤nbu(a) @(Ieme(i,a),Ieme(i,b))=⊥ then res=⊥
          else res=∀ 0<i≤<nbu(a) @(ieme(i,a),ieme(i,b))
```

Opérateurs infixés != <= >=

```
res := Fonct:(CP:(Nuple:(...):,Nuple:(...):):->Pbool:): !=(a,b)
     if ==(a,b)=⊥ then res=⊥ else if ==(a,b)=⊤
               then res=⊤ else res=!(==(a,b))
res := Fonct:(CP:(Nuple:(...):,Nuple:(...):):->Pbool:): <=(a,b)
     if >(a,b)=⊥ then res=⊥ else if >(a,b)=⊤
               then res=⊤ else res=!(>(a,b))
res := Fonct:(CP:(Nuple:(...):,Nuple:(...):):->Pbool:): >=(a,b)
     if <(a,b)=⊥ then res=⊥ else if <(a,b)=⊤
               then res=⊤ else res=!(<(a,b))
```

Le type Union:

Le type Union: dénote toutes les unions disjointes d'objets.
Un objet de type Union se compose de trois éléments :
- la référence : un Nuple dénotant les objets possibles
- un indice désignant un élément de la référence et dénotant les objets possibles de la donnée
- la donnée proprement dite.

Deux objets de type Union ne sont comparables que s'ils ont même référence et même indice.
Soient deux objets Union: de même référence nuple :

```
a=Union(i1,nuple,x) , b=Union(i2,nuple,y)
⇒ ( i1=i2 ⇒ (a≤b ⇔ x≤y) ) ∧ ( i1≠i2 ⇒ (a≰b ∧ b≰a) )
```

Les opérateurs disponibles sont tous ceux du type de la donnée.
Se référer donc au type de la donnée.

Constructeur Union :

```
res := Fct:(CP:(Pint,Nuple,type):->Union(...)):
```

Le type Set:

Le type Set: dénote tous les ensembles d'objets appartenant à un même type (qui peut être une union disjointe).
Un objet s de type Set: se compose de deux éléments :
- la référence : un objet dont le type dénote tous les objets susceptibles d'appartenir à s
- la donnée proprement dite qui est un Nuple: énumérant tous les objets (de même type) désignés par s ; ce Nuple ne peut désigner deux fois le même objet et l'ordre n'est pas significatif. La taille d'un Set n'est pas limitée.

Deux objets de type Set: ne sont comparables que s'ils ont même référence ; la relation ≤ est l'inclusion.

Un objet de type Set est mis à ⊤ dès qu'on tente de lui adjoindre ⊤.
Les opérateurs de comparaison n'y sont pas définis sauf ==, il faut utiliser << et >>. Pour effectuer ∪ ou ∩ , utiliser <<= ou >>=.

Constructeur Set :

```
res := Fonct:(Texte->Set:): Set(...)
     il suffit d'énumérer les données séparées par des virgules.
     les données doivent être correctement typées.
```

Fonction Nbu :

```
res := Fonct:(Set:(...):->Pint(1..n):): Nbu(x).
     if x=⊤ then res=⊤ else res=nbu(x)
```

Fonction In :

```
res := Fonct:(Couple(Set:(Type),Type):->Pbool:): In(s,a)
     if s=⊤ then ⊤ else res=∃a∈s
```

Lors de cette opération, un contrôle de type est effectué pour admettre les Unions de types distincts.

Opérateurs infixés + - ^ <<= >>=

```
     a+b est équivalent à a<<=b
     ^ est l'union moins l'intersection (le ou exclusif)
res := Fonct:(Couple(Set:(Type),Set(Type)):->Set(Type):): <<=(a,b)
     if a=⊤ ∨ b=⊤ then res=⊤
     else (∀ x∈a x∈res) ∧ (∀ x∈b x∈res) ∧ (∀ x∈res (x∈a ∨ x∈b))
res := Fonct:(Couple(Set:(Type),Set(Type)):->Set(Type):): >>=(a,b)
     if a=⊤ ∨ b=⊤ then res=⊤
     else (∀ (x∈a ∧ x∈b) x∈res) ∧ (∀ x∈res (x∈a ∧ x∈b))
res := Fonct:(Couple(Set:(Type),Set(Type)):->Set(Type):): ^(a,b)
     if a=⊤ ∨ b=⊤ then res=⊤
     else (∀ (x∈a ∧ x∉b) x∈res) ∧ (∀ (x∈b ∧ x∉a) x∈res) ∧
          (∀ x∈res (x∈a ∧ x∉b) ∨ (x∈b ∧ x∉a))
res := Fonct:(Couple(Set:(Type),Set(Type)):->Set(Type):): -(a,b)
     if a=⊤ ∨ b=⊤ then res=⊤
     else (∀ (x∈a ∧ x∉b) x∈res) ∧ (∀x∈res (x∈a ∧ x∉b) )
```

Opérateurs infixés << == >>

```
res := Fonct:(Couple(Set:(Type),Set(Type)):->Set(Type):): <<(a,b)
     res=(a=⊥)∨(∀ x∈a x∈b)
res := Fonct:(Couple(Set:(Type),Set(Type)):->Set(Type):): >>(a,b)
     res=(b=⊥)∨(∀ x∈b x∈a)
res := Fonct:(Couple(Set:(Type),Set(Type)):->Set(Type):): ==(a,b)
     if a=⊤ ∨ b=⊤ then res=⊤ else res=(∀ x∈a x∈b) ∨(∀ x∈b x∈a)
     (ne produira jamais ⊥)
```

Le type Seq:

Le type Seq: dénote toutes les suites d'objets appartenant à un même type (qui peut être une union disjointe). C'est une sorte de Nuple à longueur non déterminée à l'initialisation.
Un objet s de type Seq: se compose de deux éléments :

- la référence : un objet dont le type dénote tous les objets susceptibles d'appartenir à s
- la donnée proprement dite qui est un Nuple: énumérant tous les objets (de même type) désignés par s ; ce Nuple peut désigner deux fois le même objet et l'ordre est significatif. La taille d'un Seq n'est pas limitée.

Deux objets de type Seq: ne sont comparables que s'ils ont même référence ; la relation $\preceq$ est cette relation rapportée au type de la

référence et pour chaque couple d'éléments de même indice à concurrence de la longueur de la liste.
Un objet de type Seq est mis à ⊤ dès qu'on tente de lui adjoindre ⊤.
Les opérateurs de comparaison n'y sont pas définis, il faut utiliser << et >>. Pour effectuer ∪ ou ∩ , utiliser <<= ou >>=.

Constructeur Seq :

```
res := Fonct:(Texte->Seq:): Seq(...)
      il suffit d'énumérer les données séparées par des virgules.
      les données doivent être correctement typées ; l'ordre sera
conservé
```

Fonction Nbu :

```
res := Fonct:(Seq:(...):->Pint():): Nbu(x).
      if x=⊥ then res=⊤ else res=nbu(x)
```

Fonction In :

```
res := Fonct:(Couple(Seq:(Type),Type):->Pbool:): In(s,a)
      if x=⊥ then res=⊤ res=∃a∈s
```

Lors de cette opération, un contrôle de type est effectué pour admettre les Unions de types distincts.

Opérateur infixé +

```
res := Fonct:(CP:(Pstring:,Pstring:):->Pstring:): +(a,b)
      if a=⊤ ∨ b=⊤ then res=⊤
      if a=⊥ ∨ b=⊥ then res=⊥ else
            ∀ 0<i≤nbu(a) res[i]=a[i] ∧ ∀ 0<i≤nbu(b) res[nbu(a)+i]=a[i]
```

Fonction Taille

```
res := Fonct:(Pstring:->Pint:): Taille(x)
      if x=⊤ then res=⊤ else res=nbu(x)
```

Fonction Ieme

```
res := Fonct:(CP:(Pint:,Pstring:):->Pchar:): Ieme(n,s)
      if n=⊤ ∨ s=⊤ ∨ n≤0 ∨ n>nbu(s) then res=⊤
      else if n=⊥ then res=⊥ else res=s[n]
```

Fonction Subst ieme

```
res := Fonct:(CP:(Pint:,Pstring:,Pchar):->Pstring:): Subst_ieme(n,s,c)
      if n=⊤ ∨ s=⊤ ∨ c=⊤ ∨ n≤0 ∨ n>len(s) then res=⊤
      else if n=⊥ ∨ s=⊥ ∨ c=⊥ then res=⊥
            else l=nbu(s) ∧ nbu(res)=l ∧
                  ∀ 0<i<n res[i]=s[i] ∧ res[n]=c ∧ ∀ n<i≤l res[i]=s[i]
```

Opérateurs infixés <<= >>=

```
res := Fonct:(Couple(Seq:(Type),Seq(Type)):->Seq(Type):): <<=(a,b)
      if a=⊤ ∨ b=⊤ then res=⊤
      else if nbu(a)<nbu(b) then l=nbu(b) else l=nbu(a)
            ∀ 0<i≤l res[i]=a[i]<<b[i] ∧ nbu(res)=l
res := Fonct:(Couple(Seq:(Type),Seq(Type)):->Seq(Type):): >>=(a,b)
            if nbu(a)<nbu(b) then l=nbu(a) else l=nbu(b)
            ∀ 0<i≤l res[i]=a[i]>>b[i] ∧ nbu(res)=l
```

APPLICATIONS

Un système d'équations comme exposé ci-dessus est utile dans tout domaine où à chaque 'itération' il peut y avoir moyen d'améliorer les résulats acquis. On pourrait imaginer un système 'météorologique' en temps réel où chaque équation est une formule de calcul du temps qu'il fera en un endroit à partir des renseignements déjà obtenus pour les contrées voisines ainsi que quelques 'inputs' qui seraient des sondes. Mais restons modestes ...

Notre système d'équations s'applique aisément quand on a déjà un système d'équations en tête ; on pourra tester combien de consultations de table et combien de calculs effectifs sont effectués , notamment quand on répète une question ou quand on demande un calcul dont une portion a déjà été effectuée.

juste une remarque : notre langage de description d'équations est très peu 'procédural' cependant, il y a moyen d'implémenter des "while" de la manière suivante. Supposons que l'on désire garnir une donnée de type liste d'entiers depuis la frappe clavier :

```
##1,100#
Gint:max := 10 ;
Liste:(Pint:):a := droite(aa) ;
Couple:(Gint:,Liste:(Pint:)): aa
     := if gauche(aa) << max
        then Couple(++gauche(a),droite(aa)+Pint(input))
        else aa
        endif ;
```

Cette manière d'écrire montre bien comment fonctionne l'opérateur de point fixe :

1. comme aa est inconnu, a est d'abord initialisé à ⊥ c'est-à-dire Couple(1,⊥)
3. gauche(aa) donne un Gint: : l'indice en cours
4. ++gauche(aa) incrémente l'indice ou plutôt : retourne la valeur de la partie gauche de a incrémentée de 1
5. droite(aa)+Pint(input) est la liste où l'on a rajouté un élément en tête depuis le clavier
7. l'évaluation sera améliorée jusqu'à ce que droite(aa)>max.

Si l'on désire que ce soit l'opérateur qui interrompe l'introduction des nombres, on écrira :

```
Liste:(Pint:): a
     := output("affectation de a ; encore ? 0/1") ,
        if Pint(input)!=0 then a+Pint(input) else a endif.
```

Voici comment calculer le nombre de zéro dans la liste a :

```
Gint:nombre_de_0 := droite(n) ;
Couple:(Gint:,Gint:): n
     := if gauche(n)<taille(a)
        then Couple(++gauche(n) ,
                    if Ieme(a,gauche(n))==0
                    then ++droite(n) else droite(n)
                    endif
                   )
        else n
        endif ;
```

Ici, il faut faire attention : le calcul de n va demander la valeur de a ; il ne faut plus que soit posée la question "affectation de a ; encore ? 0/1" ; la valeur de a doit être stable ! Le raisonnement est simple : la valeur de toute variable sera recalculée jusqu'à ce que toutes les variables dont elle dépend sont connues ; or la valeur de n n'est pas nécessaire au calcul de a ; donc si l'on lance en premier le calcul de n, le calcul de a serait intégralement effectué à la rencontre de taille(a) puis la valeur de a ne sera plus modifiée

Notre propos initial était l'analyse statique de programmes, l'avons-nous perdue de vue ? En voici quelques exemples :

. <u>contrôle ou détermination de type</u>
ce cas sera largement traité plus loin dans ce travail

. <u>élimination de 'switch' débutant une fonction</u>
. <u>détection de récursivité terminale</u>

. <u>analyse de mode en Prolog</u>
il s'agit également d'une variante de la détermination des types : on classe les variables d'un programme Prolog en {ground,var,any}

. <u>repérage de séquences communes</u>
Si une séquence telle qu'une portion d'expression ou une suite d'instructions se révèle commune en plusieurs endroits d'un programme alors qu'aucune variable utilisée n'a été modifiée ou qu'aucun effet de bord n'est possible, on peut l'isoler et la remplacer soit par une variable soit par un appel de procédure.

. <u>détection des paramètres pouvant être passés par valeur plutôt que par nom</u>
C'est un cas particulier du précédent.

. <u>détermination du nombre de solutions à un problème</u>
on classera en {aucune,une_seule,plusieurs}

. <u>récupération d'espace mémoire alloué à des variables qui ne seront plus utilisées</u>

Choisissons d'étudier l'optimisation [2] de langages structurés "à la Pascal" :

1) <u>détermination de la portée de variables</u>

Soit un programme dans un langage qui permette de créer des variables dont le nom est partagé dans des structures imbriquées, il serait intéressant de pouvoir déterminer à tout endroit du programme quelle variable est effectivement désignée par un nom.
Soit par exemple :

```
    { x { { } x } { } { x } }
```

où x signifie qu'à cet endroit a lieu une déclaration de la variable x.

```
La structure { { { } } { } { } } peut être numérotée comme suit :
             1 2 3 2 1 4 1 5 1
```

à chaque fois qu'on entre dans une boîte on incrémente un compteur, chaque fois qu'on en sort on revient au numéro de la boîte.

---

2 : L'"optimum" existe-t-il, disons plutôt amélioration.

Voici les lieux de définition de x

```
{ { { } } { } { } }   structure du programme
 x     x      x       lieux de déclaration
```

Si l'on désigne maintenant chaque boîte par le numéro de la boîte où a été déclaré le x, on obtient :

```
{ { { } } { } { } }   structure du programme
 1 2 2 2 1 1 1 5 1    numéros de la boîte d'appartenance
```

Le problème est donc assez simple, sachant que

```
{ { { } } { } { } }   structure du programme
 1 2 3 2 1 4 1 5 1    numéros de boîtes
 x     x      x       lieux de déclaration
```

il faut trouver :

```
{ { { } } { } { } }   structure du programme
 1 2 2 2 1 1 1 5 1    numéros des x
```

Supposons que l'on ait autant de variables x que de boîtes, on peut initialiser chacune de ces variables à zéro puis lire le texte du programme ; on avançant dans le texte, on attribue des numéros aux boîtes, si l'on rencontre une déclaration de x alors on peut affecter au x de cette boîte le numéro de sa boîte. Un premier résulat sera donc

```
{ { { } } { } { } }
 1 2 3 2 1 4 1 5 1
 1 0 0 2 0 0 0 5 0
```

Il suffit maintenant d'attribuer aux x qui ont pour valeur zéro le plus grand numéro de x des boîtes plus petites ou égales.

2) <u>détermination d'espaces de non variation d'une variable</u>

Soit un programme linéaire (sans boucle), découpons ce programme en zones et demandons-nous si la valeur d'une variable est constante d'une zone à une autre. Il suffit d'attribuer à chaque zone du programme un numéro indiquant son emplacement séquentiel. Exécuter le programme en n'effectuant aucun calcul mais en attribuant à la variable lors d'une affectation le numéro de la zone où l'on se trouve. A la fin de cette exécution abstraite, la variable aura pour valeur la zone où elle aura été affectée en dernier ; elle est donc restée constante depuis lors. Si l'on désire connaître la dernière affectation pour chaque zone, considérer que la variable est un n-uplet d'autant de zones, et à chaque fois qu'elle est affectée, lui affecter sa ième partie, avec i le numéro courant de zone. Il restera encore à remplir les 'trous' en reportant à l'indice i+1 la valeur de l'indice i si en i+1 il y a 0 (valeur d'initialisation).
S'il y a des boucles, attribuer toute une zone à la boucle ; puis faire le même travail à l'intérieur de la boucle. Pour les appels de programme, même chose (ce qui permettra d'éliminer un passage de paramètre par nom).

3) <u>détermination d'expressions communes</u>

Une expression sera commune si les variables qu'elle comporte n'ont pas été modifiées, et qu'elle se retrouve en plusieurs endroits. On repérera les expressions constantes en insérant dans le programme des instructions d'affectation à une variable fictive. Si la variable est constante, l'expression est constante. Il suffit maintenant d'analyser toutes les variables fictives constantes pour en extraire celles qui ont même signification syntaxique. On pourra ainsi isoler une expression, assigner, dans le programme, sa valeur à une variable et sustituer à son calcul la désignation de la variable.

On peut généraliser à des séquences d'instructions communes que l'on remplacera par un appel de fonction.
Ce problème est assez complexe, car deux sous-expressions (ou séquences d'instructions) peuvent devenir communes après une simple permutation (qui respecte la sémantique). Des permutations peuvent être effectuées soit par application de la commutativité d'opérateurs soit par distributivité soit enfin pour les séquences d'instructions, lorsqu'aucune modification de variable n'a eu lieu. Il "suffit" donc, une fois déterminée une zone de constance des variables, de générer toutes les permutations sémantiquement possibles.
Un cas particulier sera x=x+y qui, en langage C, peut être transformé en x+=a.

4) <u>déplacement de codes hors d'une boucle</u>

Si l'on a pu déterminer une expression ou une séquence d'instructions sur des variables constantes, on peut extraire ce bloc de la boucle (pour autant bien entendu qu'il n'y ait pas d'effets de bord).

Voyons maintenant comment on pourrait mettre ces derniers exemples sous forme d'équations[3]. Chaque information sur le programme est qualifiée de l'emplacement où elle a été générée et les emplacements sont en général des blocs d'instructions ou d'expressions. A tout point du programme on dispose d'informations, si l'on envisage un bloc et plus précisément la sortie d'un bloc, on a deux solutions : ou bien une information particulère a été produite par le bloc, ou bien elle n'a pas été produite par le bloc mais avant elle et n'a pas été détruite par le bloc.
On peut déjà mettre cela sous forme d'équation :

Out(i) := Prod(i) + (In(i) - Supp(i))

avec Prod(i) : ce qui est produit par le bloc i
In(i) : ce que le bloc i a reçu
Supp(i) : ce que le bloc i a supprimé
Out(i) : ce qui sort du sort du bloc i.

Donnons un sens précis à la notion de bloc : soit un programme, découpons-le en tranches qui respectent la structure syntaxique du programme et tel qu'on obtienne une partition du programme ; attribuons à chaque bloc un identifiant. Cet ensemble de bloc a une structure : la structure syntaxique du programme ; on peut dresser un graphe qui représente les exécutions possibles du programme : les "sauts" d'un bloc à un autre.

3 : La technique ici présentée dérive des travaux en analyse de mode des programmes Prolog (confer bibliographie) ainsi que du concept d'analyse globale de flot de données que l'on trouvera dans <u>Compilateurs, principes, techniques, outils</u> de Alfred Aho, Ravi Sethi et Jeffrey Ullman.

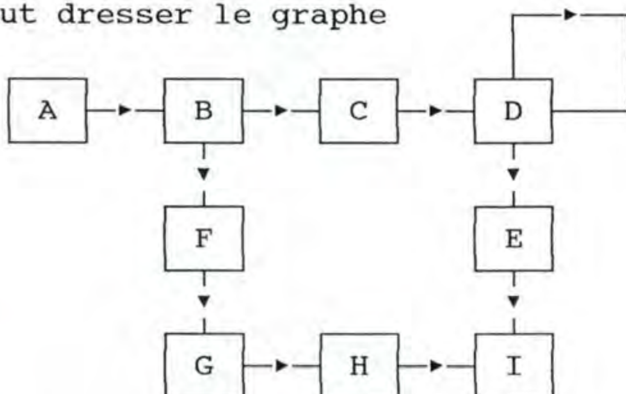
Exemple :

```
    ...
    if ...
    then ... while ... do ... enddo
    else ... perform(...) ...
    endif
    ...
```

Chaque ... désigne une séquence qui peut elle-même être partitionnée si l'on désire affiner l'analyse. Insérons des repères qui partitionnent le texte source en respectant la syntaxe (analyse dirigée par la syntaxe) :

```
    ...[A]
    if ...[B]
    then ...[C] while ...[D] do ... enddo[E]
    else ...[F] perform(...) [G] ...[H]
    endif
    ...[I]
```

On peut dresser le graphe



Le bloc G peut désigner lui-même tout un graphe ; pour l'instant, on dira que l'entrée de G c'est la sortie de F, l'entrée de H, c'est la sortie de G. Le bloc G comporte un appel de programme, on réglera son cas un peu plus tard.

Tous les exemples d'analyse exposés plus haut tournent autour de la notion de d'affectation de valeur à une variable ; l'information utile est "la valeur actuelle de cette valeur est stable depuis tel endroit du programme" ; le codomaine de nos équations sera donc un ensemble d'identifiants d'assignation de variable. Toutefois, comme nous allons devoir exécuter le programme et que celui-ci comportera des branchements, il arrivera qu'à un point du programme convergent deux exécutions distinctes faisant suite à deux affectations distinctes de la même variable. Dans ce cas, on ne pourra plus dire que la valeur est "stable" mais simplement qu'elle est stable depuis l'une OU l'autre des affectations.

Pour simplifier, nous allons dire qu'existent en certains points du programme des <u>définitions</u> de variable. Chaque définition aura un identifiant composé de la variable et de la boîte où est effectuée la définition ; signifiant par là que cette variable reçoit (probablement) une nouvelle valeur en cet endroit et que donc les définitions précédentes y sont (probablement) écrasées. "Probablement" car on peut toujours écrire a:=a !

Le codomaine de nos variables d'équations est un ensemble de définitions. La formule :

Out(i) := Prod(i) + (In(i) - Supp(i))

est à lire comme suit :
i identifie une boîte
In(i) : ensemble des définitions reçues par cette boîte
ça ne veut pas dire que cette définition soit d'application mais simplement qu'il existe un chemin dans le graphe exposé ci-dessus entre le lieu de création de cette définition et i.
Supp(i) : ensemble des définitions qui sont à coup sûr détruites au sein de i
Prod(i) : ensemble des définitions qui sont probablement produites au sein de i
Out(i) : ensemble des définitions en sortie de cette boîte
ça ne veut pas dire que la définition sera effectivement produite lors d'une exécution, mais que c'est possible : on a pu passer par là.

La définition de Prod énonce un "probable", c'est une surestimation ; on verra que dans ce cas, il vaut mieux surestimer que sous-estimer ; par contre la définition de Supp est une sous-estimation.

Ci-dessous, on utilisera $\cap$ pour désigner >>=, c'est-à-dire l'intersection de deux ensembles, c'est plus commode ; rappels : + est identique à <<= qui est l'union ; - est la différence de deux ensembles.

Le système d'équations à produire, devra être une liste telle que pour chaque bloc, on ait :
Out(i) := Prod(i) + (In(i) - Supp(i))
Prod(i) := ...
In(i) := ...
Supp(i) := ...
Il faut maintenant générer les ... ; prenons quelques exemples :

Soit le bloc i qui comporte a:=b+c

On peut dire que Prod(i)={(a,i)}
Supp(i)={(a,x) $\forall x \wedge x \neq i$}
Out(i) := {(a,i)} + (In(i) - {(a,x) $\forall x \wedge x \neq i$})
cette dernière équation, n'est pas à générer, elle devra l'être automatiquement pour chaque boîte.
On devra seulement générer les équations :
Prod(i):={(a,i)}
Supp(i):={(a,x) $\forall x \wedge x \neq i$}
In(i) est supposé connu.

Soit le bloc i qui est la séquence des blocs i_1 et i_2 : i_1;i_2

on suppose que l'on dispose déjà des équations
Out(i_x) := Prod(i_x) + (In(i_x) - Supp(i_x))
Prod(i_x) := ...
Supp(i_x) := ...
pour i_1 et i_2

i_1 reçoit en entrée ce que reçoit i, donc :
In(i_1) := In(i) ;
ce qui entre dans i_2, c'est qui sort de i_1, donc
In(i_2) := Out(i_1)
ce qui sort de i, c'est ce qui sort de i_2, donc
Out(i) := Out(i_2) mais cette ligne n'est pas à écrire !!
ce qui est produit par i, c'est ce qui a été produit par i_1 et n'a pas été détruit par i_2 et ce qui a été produit par i_2

Prod(i) := Prod(i_2) + (Prod(i_1)-Supp(i_2))
ce qui est supprimé par i, c'est ce qui a été supprimé par i_2 et ce qui a été supprimé par i_1 sauf ce qui a été restauré par i_2
Supp(i) := Supp(i_2) + (Supp(i_1)-Prod(i_2))

Soit le bloc i qui se compose de : if i_0 then i_1 else i_2

On traitera d'abord ce bloc comme [i_0 ; i_3]
On peut en effet considérer que le test est un bloc en soi.
Si le test ne comporte aucune assignation comme dans if (a=b)
ce bloc i_0 aura pour formule Prod(.)={} Supp(.)={}
Traitons donc i=[then i_1 else i_2]

Encore une fois, on suppose que l'on dispose déjà des équations pour i_1 et i_2.
Tout ce qui est produit par i est ce qui a été produit par i_1 <u>ou</u> par i_2 :
Prod(i) := Prod(i_1) + Prod(i_2)
Tout ce qui a été détruit par i est ce qui a été détruit par i_1 <u>et</u> i_2
Supp(i) := Supp(i_1) $\cap$ Supp(i_2)
Il faut encore :
In(i_1) := In(i) ; In(i_2) = In(i)

Soit le bloc i qui se compose de : while i_1 do i_2 enddo

On considérera ici i_3 comme étant la séquence i_1;i_2
On a tout simplement :
Prod(i) := Prod(i_3)
Supp(i) := Supp(i_3)
Pour le In(i_3), c'est plus subtil :
In(i_3) := In(i) + Out(i_3)
et l'on sait que Out(i_3) sera calculé par
Out(i_3) := Prod(i_3) + (In(i_3) - Supp(i_3))

Cas particuliers :

la séquence i = [i_1 ; goto i_1] sera traitée comme un [while [] do i_1 enddo]
la séquence i = [do i_1 while i_2] sera traitée comme un [while i_2 do i_1 enddo]

L'implémentation d'un tel système d'équations n'est pas trop compliquée, on assigne à chaque boîte un numéro , puis l'on crée un type par défaut
#Set:(Pint:):#
et il suffit décrire tous les :
```
in_1 := ...
prod_1 := ...
supp_1 := ...
out_1 := prod_1 + ( in_1 - supp_1 ) ;
```
ce sera long et fastidieux, mais on a rien sans mal ...

<u>analyse de mode de programmes Prolog</u>

La technique exposée ci-dessus peut être appliquée à un langage déclaratif comme Prolog en vue de déterminer certaines propriétés de clauses. On n'entrera pas dans les détails pour ne pas avoir à exposer l'unification, les substitutions, les build-in, les foncteurs etc ; on donnera simplement des indications de principe. On supposera que la fonction lub calcule la propriété désirée.

. le i de In(i), Out(i) etc désigne ici
soit soit un prédicat
soit une clause d'un prédicat
soit un sous-but d'une clause d'un prédicat
désignons par PA le prédicat A,
par CA_i une clause du prédicat A
désignons aussi par B_jCA_i un sous-but apparaissant dans CA_i

. On acceptera l'écriture $PA=B_jCX_i$ pour signifier que
le sous-but B_j de la clause C i du prédicat X
porte sur le prédicat PA

. <u>Le In de chaque clause</u> est le In du prédicat :
$In(CA_i)$:= In(PA)

. <u>Le Out de chaque clause</u> est le lub des Out des sous-buts de la clause
$Out(CA_i)$:= $lub(Out(B_jCA_i))$

. <u>Le In de chaque prédicat</u> est le lub des In de tous les sous-buts correspondant à ce prédicat dans toute clause
In(PA) := $lub(In(B_iCX_i))$
tel que $PA=B_jCX_i$

. <u>Le Out de chaque prédicat</u> est le lub des Out des clauses de ce prédicat
Out(PA) := $lub(Out(CA_i))$

. <u>Le In de chaque sous-but</u> est le In de la clause où il se trouve
$In(B_jCA_i)$:= $In(CA_i)$

. <u>Le Out de chaque sous-but</u> est le Out du prédicat qui lui correspond
$Out(B_jCX_i)$:= Out(PA)
tel que $PA=B_jCX_i$

On n'a pas abordé ci-dessus les "cas de base" que seront en général les les foncteurs ; lors d'un cas de base la formule "Le Out de chaque sous-but" est à modifier.
Si le sujet de l'analyse est la directionnalité d'un prédicat, les cas de base seront les sous-buts X=constante ou les build-in X=Y.
Si le sujet de l'analyse est 'le nombre de solutions possibles' les cas de base seront également ces X=constante qui donnera 1, alors que l'opération de lub est, par exemple, une addition simplifiée : {0,1,beaucoup}.

Cette application demande l'implémentation d'un nouveau domaine et d'opérateurs adéquats ; on n'exposera pas ici ces problèmes de quincaillerie, le code du programme comporte suffisamment de commentaires pour programmeur C orienté objet.

SEMANTIQUE DU PROGRAMME

On exposera ici la sémantique du programme en supposant que les analyses lexicales et syntaxiques ont pu produire une séquence d'équations abstraites syntaxiquement correctes. Bien que le programme effectue une analyse lexicale (extraction des symboles du langage tels que ;.(,)[|] et reconnaissance des constantes et identificateurs de types, fonctions, opérateurs ou variables), ainsi qu'une analyse syntaxique (articulation correcte de tous ces symboles), on considérera ici qu'il s'agit d'un problème 'classique' qu'il n'est pas nécessaire d'exposer.

L'exposé sera fait comme suit :

- définitions ;
- sémantique de l'exécution d'un système d'équations sémantiquement correct ;
- sémantique de l'analyse sémantique d'un système d'équations syntaxiquement correctes ;

Nous verrons plus loin pourquoi la sémantique du programme d'analyse des types est exposée après celle du programme d'exécution alors que ce dernier en a besoin.

L'exposé de la sémantique du programme demande l'usage d'un métalangage ; ce métalangage sera considéré comme évident (faut bien un point de départ). Notre "métalangage" articulera des opérations qui devront être <u>continues</u> sur des domaines <u>inductifs</u>.

Une opération sera définie en deux temps :

1. signature ou domaine d'appartenance de la fonction ;
2. définition en compréhension des couples
   généralement par induction sur la structure de la donnée

Les couples seront définis à partir des opérateurs classiques que sont :
- {} pour définir un ensemble par compréhension ou énumération
- $\in$ $\cup$ $\cap$ $\setminus$ de la théorie des ensembles
- $\forall$ $\exists$ $=$ $\neq$ $\wedge$ $\vee$ $\Leftrightarrow$ $\Rightarrow$ "si alors sinon" de la logique des prédicats
- μ opérateur de point fixe
- λ opérateur de lambda calcul.

Le symbole $\triangleq$ signifiera "est défini par".
On emploiera encore l'opérateur de substitution [./.]
par exemple dans f[x/y] qui signifiera
f mais avec x dans f remplacé par y :

$$f[x/y]=g \Leftrightarrow \forall a\in f \;(\; a=x \Rightarrow y\in g \;) \wedge (\; a\neq x \Rightarrow a\in g \;)$$
$$\forall a\in g \;(\; a=y \Rightarrow x\in f \;) \wedge (\; a\neq y \Rightarrow a\in f)$$

Certains domaines seront clairement définis (comme le domaines des types) ; d'autres non, ce sera le cas de domaines provenant de l'analyse syntaxique. Les données provenant de l'analyse syntaxique ou sémantique seront encadrées par []. Entre les [], on trouvera des caractères ainsi que des mots entre <> ; les mots entre <> désigneront des données de la syntaxe abstraite alors que les caractères n'auront pas de signification : ils ne seront présents que pour aider la lecture ; ce seront des symboles issus de la syntaxe concrète (confer plus loin l'exposé de la syntaxe, des règles de grammaire).

EQUATION :: ensemble des équations syntaxiquement correctes.

On désignera une équation par

```
     [<tête>:=<expression>]
         t        e
```

signifiant par là que l'analyseur syntaxique a isolé t et e.
On n'a pas à définir ici ce qu'est une EQUATION ; c'est le problème de l'analyse syntaxique ; semblablement on acceptera les ensembles :

TETE EXPRESSION PARAMETRE

de telle sorte que lorsqu'une occurrence d'un objet de l'un de ces types est présenté par l'analyseur syntaxique, on puisse obtenir immédiatement sa décomposition par exemple dans : [<tete>]=[<type><nom>(<paramètres>)]

SEQ[DOMAINE]

Soit D un domaine (ensemble de valeurs) quelconque, définissons SEQ[D] comme étant l'ensemble des séquences d'objets du domaine D de par les définitions abstraites suivantes (qui sont ici identiques à celles de la notion de PILE car on ne se préoccupe pas du comment les séquences ont été garnies) :

Fonctions :

```
     Empiler : SEQ[D] x D -> SEQ[D]
          ss = Empiler(s,d)
          * ss est la séquence s où l'on a ajouté d∈D
     Dépiler : SEQ[D] -> SEQ[D]
          ss = Dépiler(s)
          * ss est la séquence s où l'on a retiré le premier élément
     Tête : SEQ[D] -> D
          d = Tête(s)
          * d est le premier élément de la séquence s
     Svide : -> SEQ[D]
          s = Svide()
          * s est une séquence vide
     Svide? : SEQ[D] -> BOOL
          Pvide?(s)
          * teste si la séquence s est vide
```

Equations

```
     Svide?(Svide()) = vrai
     Svide?(Empiler(s,d)) = faux
     Dépiler(Empiler(s,d)) = s
     Dépiler(Svide()) = erreur
     Tête(Empiler(s,d)) = d
     Tête(Pvide()) = erreur
```

SEQ[EQUATION] :: ensemble des séquences d'équations

Supposons que le résultat de l'analyse syntaxique est un objet du type SEQ[EQUATION] :

```
Si s∈SEQ[EQUATION] alors ou bien s=[]     : la séquence est vide
                         ou bien s=[e|s']  : e est la séquence de tête de s
```

IDENTIFICATEUR :: ensemble des identificateurs syntaxiquement corrects.

Cet ensemble pourra, mais ce n'est pas indispenable, être muni de la relation d'ordre partielle lexicographique.

TYPE :: ensemble des types.

Cet ensemble est défini récursivement par

TYPE = { Nul:,Top: } ∪
{ Pbool:,Pint:,Preal:,Pchar:,Pstring: } ∪
{ Gint:(a,b): | $a \leq b$ } ∪
{ Lint:(a,b): | $a \leq b$ } ∪
{ Borne:(t): | t ∈ TYPE } ∪
{ Seq:(t): | t ∈ TYPE } ∪
{ Set:(t): | t ∈ TYPE } ∪
{ Couple:(t1,t2): | t1,t2 ∈ TYPE } ∪
{ Fct:(t1->t2): | t1,t2 ∈ TYPE } ∪
{ $Nuple_n$: | $0<n \wedge$
$Nuple_n$: = { Nuple:($t_1,t_2,..t_n$): | $\forall\ 0<i\leq n\ t_i \in$ TYPE } }
{ $Union_n$: | $0<n \wedge$
$Union_n$: = { Union:($t_1,t_2,..t_n$): | $\forall\ 0<i\leq n\ t_i \in$ TYPE } }

Remarques : Couple:(t1,t2): est un type, Couple(a,b) est une occurrence de couple. On a donc la relation Couple(a,b) ∈ Couple:(t1,t2):
Fct:(t1,t2): est une partie de Couple:(t1,t2): mais leurs éléments ne seront pas considérés comme comparables ; de même pour Nuple:(t1,t2):.
On pourrait à la rigueur accepter les types suivants $Nuple_0$: = $Union_0$: = Nul:

Chaque élément de TYPE est un treillis, rappelons :

Domaine minimal (qui n'a qu'un élément) :
Nul: = $\{\bot_{Nul:}\}$

Domaine maximal (qui n'a qu'un élément) :
Top: = $\{\top_{Top:}\}$

Domaines primitifs :
(on supposera integer,real,char et string déjà définis)
Pbool: = $\{\bot_{Pbool:}$,TRUE,FALSE,$\top_{Pbool}\}$
avec $\bot_{Pbool:} \prec \top_{Pbool:} \wedge \forall\ x \in$ boolean $\bot_{Pbool:} \preceq x \wedge x \preceq \top_{Pbool}$
Pint: = $\{\bot_{Pint:}, \top_{Pint}\} \cup \{x | x \in$integer$\}$
avec $\bot_{Pint:} \prec \top_{Pint} \wedge \forall\ x \in$ integer $\bot_{Pint:} \preceq x \wedge x \preceq \top_{Pint}$
Preal: = $\{\bot_{Preal:}, \top_{Preal}\} \cup \{x | x \in$real$\}$
avec $\bot_{Preal:} \prec \top_{Preal} \wedge \forall\ x \in$ real $\bot_{Preal:} \preceq x \wedge x \preceq \top_{Preal}$
Pchar: = $\{\bot_{Pchar:}, \top_{Pchar}\} \cup \{x | x \in$char$\}$
avec $\bot_{Pchar:} \prec \top_{Pchar} \wedge \forall\ x \in$ char $\bot_{Pchar:} \preceq x \wedge x \preceq \top_{Pchar}$
Pstr: = $\{\bot_{Pstr:}, \top_{Pstr}\} \cup \{x | x \in$string$\}$
avec $\bot_{Pstr:} \prec \top_{Pstr} \wedge \forall\ x \in$ string $\bot_{Pstr:} \preceq x \wedge x \preceq \top_{Pstr}$

Domaine des bornes :

$\text{Borne:}(t)\text{:} = \{(a,b) \mid a,b \in t\}$

avec $(a,b) \preceq_{\text{Borne:}(t)\text{:}} (x,y) \Leftrightarrow x \preceq_t a \wedge b \preceq_t y$

$\forall a,b \in t \; b \prec_t a \Rightarrow (a,b) = \bot_{\text{Borne:}(t)\text{:}}$

$(\bot_t, \bot_t) \succ \bot_{\text{Borne:}(t)\text{:}}$

$\top_{\text{Borne:}(t)\text{:}} = (\bot_t, \top_t)$

Domaines sur intervalle d'entiers

$\text{Gint:}(\min,\max)\text{:} = \{x \mid \min \le x \le \max\}$

$\text{Lint:}(\min,\max)\text{:} = \{x \mid \min \le x \le \max\}$

Domaines des fonctions composées récursivement :

$\text{Fct:}(t_1,t_2)\text{:} = \{f \mid f = \{(a,b) \mid a \in t_1 \wedge b \in_{t2}$
$\wedge \; \forall \; (a,b),(x,y) \in f \;\; a = x \Rightarrow b = y \;\}$

avec $\forall \; f,g \in \text{Fct:}(t_1,t_2)\text{:}$

$f \preceq_{\text{Fct:}(t1\to t2)\text{:}} g \Leftrightarrow \forall (a,b) \in f \; \exists \; (a,c) \in g \wedge b \preceq c$

$\bot_{\text{Fct:}(t1\to t2)\text{:}} \in \text{Fct:}(t_1,t_2)\text{:} \mid \forall x \in t_1 \;\; \bot_{\text{Fct:}(t1\to t2)\text{:}}(x) = \bot_{t1}$

$\top_{\text{Fct:}(t1\to t2)\text{:}} \in \text{Fct:}(t_1,t_2)\text{:} \mid \forall x \in t_1 \;\; \top_{\text{Fct:}(t1\to t2)\text{:}}(x) = \top_{t1}$

$\bot_{\text{Fct:}(t1\to t2)\text{:}} \preceq_{\text{Fct:}(t1\to t2)\text{:}} f \preceq_{\text{Fct:}(t1\to t2)\text{:}} \top_{\text{Fct:}(t1\to t2)\text{:}}$

Domaines des uples composés récursivement :

$\text{Couple:}(t_1,t_2)\text{:} = \{(a,b) \mid a \in t_1 \wedge b \in t_2\}$

avec $\forall \; (a,b),(x,y) \in \text{Couple}(t_1,t_2)$

$(a,b) \preceq_{\text{Couple}} (x,y) \Leftrightarrow a \preceq_{t1} x \wedge b \preceq_{t2} y$

$\bot_{\text{Couple:}(t1,t2)\text{:}} = (\bot_{t1}, \bot_{t2}) \preceq (a,b)$

$a = \top_{t1} \vee b = \top_{t2} \Rightarrow (a,b) = \top_{\text{Couple:}(t1,t2)\text{:}} = (\top_{t1}, \top_{t2})$

$\text{Nuple:}(t_1,t_2,..t_n)\text{:} = \{(x_1,x_2,...x_n) \mid x_i \in t_i\}$

avec $(x_1,x_2,...x_n) \preceq (y_1,y_2,...y_n)$

$\Leftrightarrow \forall \; 0<i \le n \;\; x_i \preceq_{ti} y_i$

pour $\bot$ et $\top$, il suffit d'étendre la définition donnée pour $\text{Couple}(t_1,t_2)$, avec par exemple :

$\forall \; 0<i \le n \; (\; x_i = \top_{ti} \Rightarrow \forall \; 0<j \le n \;\; x_j = \top_{tj} \;)$

Unions disjointes de domaines

$\text{Union:}(t_1,t_2,..t_n)\text{:} = \{(i,x) \mid 0<i \le n \wedge x \in t_i\} \cup \{\bot_{\text{Union}}, \top_{\text{Union}}\}$

Ensembles et séquences

$Set:(t): = \{\{x|x \in t\}\}$

avec $a \preceq_{Set:(t):} b \iff \forall x \in a\ x \in b$

et $\forall s \in Set:(t):\ \top_t \in s \Rightarrow s=\{\top_t\}$

$Seq:(t): = \{\{\} \cup \{(x,s)\ |\ x \in t \wedge s \in Seq:(t):\}\}$

avec $a \preceq_{Seq:(t):} b$

$\iff (\ a=\{\}\ ) \vee$

$(\ a=(x,y) \wedge b=(x,z) \wedge y \preceq_{Seq:(t):} z\ ) \vee$

$(\ a=(x,y) \wedge b=(u,v) \wedge x \not\succeq_{Seq:(t):} u\ )$

et $\forall s \in Seq:(t):\ s=(x,y) \wedge x=\top_t \Rightarrow y=\{\}$

Rappelons que les fonctions glb (ou borne inférieure, ou >>=) et lub (ou borne supérieure, ou <<=) sont définies sur chacun des domaines ci-dessus et que ces deux fonctions sont totales et monotones :

Remarque : on n'exposera ici que les fonctions lub ; les fonctions glb se conçoivent aisément à partir de lub. On ne démontrera pas non plus que chaque lub ou glb est fonction monotone, c'est assez évident.

Pour tout 'domaine primitif' ici désigné par D :

$\forall a,b \in D$ on a $a \ll_D b = c\ |\ a \preceq_D c \wedge a \preceq_D c\ \ \wedge \forall d \in D\ |\ a \preceq_D d \wedge b \preceq_D d \Rightarrow c \preceq_D a$

Soit un type D=Borne:(T): avec $T \in TYPE$

$\forall (a,b),(x,y) \in D$ on a $(a,b) \ll_D (x,y) = (v,w)$

$|\ (a,b) \preceq_D (v,w) \wedge (x,y) \preceq_D (v,w)$

$\wedge \forall (t,p) \in D\ |\ (a,b) \preceq_D (t,p) \wedge (x,y) \preceq_D (t,p) \Rightarrow (v,w) \preceq_D (t,p)$

On a donc toujours :

$\forall (a,b) \in D\ \ \bot_D=(\bot_{D1},\bot_{D2}) \preceq (a,b) \preceq \top_D=(\top_{D1},\top_{D2})$

Soit un type D=Couple:(D_1,D_2):

$\forall (a,b),(x,y) \in D$ on a $(a,b) \ll_D (x,y) = (v,w)$

$|\ (a,b) \preceq_D (v,w) \wedge (x,y) \preceq_D (v,w)$

$\wedge \forall (t,p) \in D\ |\ (a,b) \preceq_D (t,p) \wedge (x,y) \preceq_D (t,p) \Rightarrow (v,w) \preceq_D (t,p)$

On a donc toujours :

$\forall (a,b) \in D\ \ \bot_D=(\bot_{D1},\bot_{D2}) \preceq (a,b) \preceq \top_D=(\top_{D1},\top_{D2})$

Soit un type $D \in Nuple_n$: et D=Nuple:$(D_1,D_2,\ldots D_n)$

si l'on désigne par x_i le ième élément du nuple

$x \in$ Nuple:$(D_1,D_2,\ldots D_n)$

$\forall x,y \in D$ on a $x \ll y = z$

$|\ (\forall 0<i \le n\ x_i \preceq z_i \wedge y_i \preceq z_i) \wedge$

$\forall\ e \in D\ |\ (\forall 0<i \le n\ x_i \preceq e_i \wedge y_i \preceq e_i) \Rightarrow z \preceq e$

On a donc toujours :

$\forall x \in D\ \ (\bot_{D1},\bot_{D2},\ldots,\bot_{Dn}) \preceq x \preceq (\top_{D1},\top_{D2},\ldots,\top_{Dn})$

Soit un type $D \in Union_n$: et $D = Union:(D_1, D_2, \ldots D_n)$

remarquez ci-dessus la présence des deux éléments particuliers que sont $\bot_{Union:(\ldots)}$ et $\top_{Union:(\ldots)}$, ils sont nécessaires à assurer la totalité des fonctions lub et glb lors de la comparaison de deux objets de type distincts

$\forall x,y \in D$ on a $x \ll y = z$

tel que soit $x=(i_x,x') \wedge y=(i_y,y')$

si $i_x = i_y$

alors $z=(i_x, x' \ll y')$

sinon $z=\top_{Union:(..)}$

Soit un type $D=Fct:(D_1 \text{->} D_2)$: | $D_1, D_2 \in$ TYPE

on sait que $\bot_D$ est la fonction constante qui retourne $\bot_{D2}$

et que $\top_D$ est la fonction constante qui retourne $\top_{D2}$

$\forall f,g \in D$ on a $f \ll g = h$

tel que $\forall x \in D_1$ $h(x)=f(x) \ll g(x)$

Soit un type $D=Seq(T)$ avec $T \in$ TYPE

$\forall x,y \in D$ on a $x \ll y = z$

tel que si $x \preceq y$ alors $z=y$ sinon $z=x$

Remarquons que $\bot_D=\{\}$ avec $\bot_D \preceq (\bot_T,\{\}) \preceq (\bot_T,x)$

et que $\top_D$ est tout simplement $\{\top_T\}$

Soit un type $D=Set(T)$ avec $T \in$ TYPE

$\forall x,y \in D$ on a $x \ll y = z$

tel que si $x \preceq y$ alors $z=y$ sinon $z=x$

Remarquons que $\bot_D=\{\}$ avec $\bot_D \preceq (\bot_T,\{\}) \preceq (\bot_T,x)$

et que $\top_D$ est tout simplement $(\top_T,\{\})$

L'ensemble (TYPE, $\preceq_{TYPE}$) est un treillis par le fait que :

(dorénavant, nous n'indicerons $\preceq$ ou $\perp$ de son type que lorsque c'est utile à l'exposé ; ci-dessus, nous avons fait les distinctions pour bien montrer qu'il s'agit d'opérateurs ou valeurs bien distincts)

$\perp_{TYPE}$=Nul: $\top_{TYPE}$=Top:

la relation partielle $\preceq_{TYPE}$ est définie par

$\forall$ t $\in$ TYPE $\perp \preceq t \preceq \top$ $\wedge$
$\forall$ a,b$\in$TYPE $\wedge$ a$\preceq$b => Borne:(a): $\preceq$ Borne:(b): $\wedge$
Set:(a): $\preceq$ Set:(b): $\wedge$ Seq:(a): $\preceq$ Seq:(b): $\wedge$
$\forall$ Couple:(a,b): , Couple:(x,y):
a$\preceq$x $\wedge$ b$\preceq$y <=> Couple:(a,b): $\preceq$ Couple:(x,y):
$\forall$ a,b $\in$ $Nuple_n$ | a=Nuple:(t_1,t_2,..t_n): , b=Nuple:(t'_1,t'_2,..t'_n):
a$\preceq$b <=> $\forall 0<i\leq n$ $t_i \preceq t'_i$
$\forall$ a,b $\in$ $Union_n$ | a=Union:(t_1,t_2,..t_n): , b=Union:(t'_1,t'_2,..t'_n):
a$\preceq$b <=> $\forall 0<i\leq n$ $t_i \preceq t'_i$
$\forall$ a$\notin$ Fct(:(t_{11}->t_{12}) $\wedge$ b$\notin$ Fct(:(t_{21}->t_{22})
a$\preceq$b <=> $t_{11} \preceq t_{21}$ $\wedge$ $t_{12} \preceq t_{22}$

La relation $\preceq$ ainsi définie, on obtient l'opérateur lub qui nous permettra de définir le type d'un objet par approximations successives. Il est à remarquer que nous n'avons pas introduit de 'constructeur' de type du genre de
$\forall$ a$\in$TYPE a$\preceq$Union:(t_1,t_2,..t_n): si $\exists 0<i\leq n$ | t_i=a
ou encore
$\forall$ a$\in$TYPE a$\preceq$Union:(t_1,t_2,..t_n,a): avec
Union:(t_1,t_2,..t_n): $\preceq$ Union:(t_1,t_2,..t_n,t_{n+1}):
Car l'opérateur lub n'aurait plus été commutatif.

On verra plus tard que lorsque nous chercherons à déterminer le type d'une variable par approximations successives, par exemple dans le cas suivant :

```
Pint:a := ... ;
Preal:b := ... ;
c := f(a) ;
d := f(b) ;
f(x) :=
```

où l'on découvre d'abord que le paramètre de f est Pint: puis qu'il faut effectuer lub(Pint:,Preal), l'unique solution est $\top$.

<u>VALEUR</u> :: ensemble des valeurs possibles

Valeur est défini comme l'union disjointe des éléments de chaque type : VALEUR = { x | x$\in$t $\wedge$ t$\in$TYPE }
En cas de confusion possible, on pourra toujours écrire :
VALEUR = { (x,t) | x$\in$t $\wedge$ t$\in$TYPE }

VALEUR est un treillis de par :

$\forall$ v$\in$VALEUR $\perp_{Nul:} \preceq_{VALEUR}$ v

$\forall$ v$\in$VALEUR v $\preceq_{VALEUR} \top_{Top:}$

$\forall$ x,y $\in$VALEUR

si x et y sont de types distincts alors lub(x,y)=$\top_{Top:}$

ou $x \in t_1 \wedge y \in t_2 \wedge t_1 \neq t_2$ => x<<y=$\top$

$\forall$ x,y$\in$t $\wedge$ t$\in$TYPE $x \preceq_{TYPE} y$ => $x \preceq_{VALEUR} y$

on définira alors la fonction lub : VALEUR x VALEUR -> VALEUR
telle que $\forall$ x,y$\in$t $\wedge$ t$\in$TYPE
z=lub(x,y) tel que
z$\in$t $\wedge$ x$\preceq$z $\wedge$ y$\preceq$z $\wedge$
($\forall$ w$\in$t | x$\preceq$w $\wedge$ y$\preceq$w => z$\preceq$w)

Cette fonction lub est partielle puisque lub(Pint:a,Preal:b) n'a pas de solution.
cas particulier si $x <<_{Union:(...)} y = \top_{Union:}$ alors lub(x,y)=$\top_{Top:}$

<u>FONCTION</u> :: ensemble des fonctions

FONCTION est l'ensemble des nuplets (id,t_1,t_2,T)

avec id $\in$ IDENTIFICATEUR l'identificateur de la fonction
$t_1 \in$ TYPE le type du domaine de la fonction
$t_2 \in$ TYPE le type du codomaine de la fonction
T = { (x,y) | $x \in t_1 \wedge y \in t_2$ } la table des valeurs
avec $\forall x \in t_1 \; \exists y \in t_2$ | (x,y)$\in$T
$\forall$ (a,b) $\in$ T (x=a => y=b) $\wedge$
$\forall$ (x,y),(a,b) $\in$ T (x$\preceq$a => y$\preceq$b) }

Autrement dit, FONCTION est l'ensemble des fonctions <u>totales</u> et <u>monotones</u> dans nos types.

Ce type de données sera celui des fonctions et opérateurs prédéfinis (offerts dans le "lexique") et qu'on appelera "oracles" ainsi que celui des variables paramétrées dans le système d'équation. On verra que la monotonie des variables paramétrées sera assurée par le fait que toute insertion dans la table effectuera un lub sur les valeurs déjà connue.

On peut encore définir FONCTION comme l'ensemble des nuplets (id,t_1,t_2,v_1,v_2) avec $v_1 \in t_1$ et $v_2 \in t_2$; ce qu'on n'hésitera pas à faire par simplification.

<u>ENVIRONNEMENT</u> :: ensemble de fonctions compatibles

ENVIRONNEMENT est une partie de FONCTION telle que tous ses éléments respectent la condition suivante :

e $\in$ ENVIRONNEMENT => $\forall$ f$\in$e f$\in$FONCTION $\wedge$ f=(id,t1,t2,T)
$\forall$ g$\in$e $\wedge$ g=(id',t1',t2',T') $\wedge$
(id=id' => t1$\neq$t1')

Autrement dit, toutes les fonctions qui ont même identificateur se distinguent de par le type de leur domaine. C'est nécessaire pour l'analyse des types, de manière à pouvoir déterminer la fonction à exécuter en fonction des paramètres. Signalons que le type du codomaine ne permet pas de distinguer deux fonctions.

Par simplication, on dira aussi qu'un ENVIRONNEMENT est un ensemble de 5-nuplets (id,t_1,t_2,v_1,v_2).

La définition ci-dessus est d'application lors de l'analyse sémantique ; mais une fois l'analyse sémantique terminée, tous les types ont été définis et donc également ceux des fonctions. On considérera donc par la suite que les noms des fonctions de l'environnement ne sont pas surdéfinis quitte à ce que l'analyseur sémantique doive fournir un nouvel environnement sans nom de fonction partagé.

RESULTAT :: ensemble de solutions compatibles

RESULTAT est une partie de FONCTION telle que tous ses éléments respectent la condition suivante :

Soit e un ENVIRONNEMENT de référence

$$r \in RESULTAT \Rightarrow \forall x \in r \quad x \in FONCTION \wedge x=(id,t1,t2,T)$$
$$\nexists g \in e \mid g=(id,_,_,_) \wedge$$
$$\forall y \in r \; y=(id,_,_,_) \; x=y$$

En français : un nom de variable identifie la variable et n'est partagé par aucune fonction de l'environnement de référence.

Par simplification, on dira aussi qu'un RESULTAT est un ensemble de 5-uplets (id,t_1,t_2,v_1,v_2) avec $v_1 \in t_1$ et $v_2 \in t_2$.

On constate qu'une table de valeurs calculées doit correspondre à une fonction totale et monotone.

On peut dès à présent donner quelques indications quant à la sémantique du programme, cela aidera à la compréhension de la suite : au fur et à mesure de la lecture des équations fournies par l'analyseur sémantique qui a pu déterminer les types de chaque variable, on complète une donnée de type RESULTAT ; pour chaque équation, la table est calculée mais comme toutes les équations n'ont pas encore été lues, certaines variables sont inconnues et alors estimées à $\bot$. Une fois que toutes les équations sont lues il faudra recommencer le même travail jusqu'à ce que la donnée de type RESULTAT ne puisse être améliorée.

Les objets de type RESULTAT seront 'sémantiquement' munis de la fonction suivante qui assure une insertion monotone en table :

$$+ : RESOLU \times IDENTIFICATEUR \times TYPE \times TYPE \times VALEUR \times VALEUR \rightarrow RESOLU : (r_1,id,td,tc,vd,vc,q) \rightarrow r_2$$

$r_1+(id,td,tc,vd,vc)=r_2$ tel que

si $\exists\ (id,td,tc,vd,x) \in r_1$
alors $r_3=r_1 \setminus \{(id,td,tc,vd,x)\} \cup \{(id,td,tc,vd,lub(vc,x))\}$
sinon $r_3=r_1 \cup \{(id,td,tc,vd,x)\}$ $\wedge$
$r_2 = \{(id',td',tc',vd',vc') \mid$
$(\ (id',td',tc',vd',vc') \in r_3 \wedge id' \neq id\ ) \vee$
$(\ (id',td',tc',vd',y) \in r_3 \wedge id'=id \wedge$
si $vd \leq vd'$
alors $vc'=lub(y,x)$
sinon $vc'=y$ $)\}$

Sémantique du programme de calcul d'un système d'équations sémantiquement correct

Sémantiquement correct signifie que
- les types sont partout connus, nulle part n'apparaît Nul: (sauf domaine de variable simple) ni Top:
- chaque équation est exécutable sans erreur : toute fonction recevra le nombre de valeurs qu'elle nécessite et ces valeurs seront du bon type
- toutes les variables du type fonction sont continues.

Le programme calcule le plus petit point fixe de la transformation suivante :

```
P : SEQ[EQUATION] x ENVIRONNEMENT x RESULTAT ->
    SEQ[EQUATION] x ENVIRONNEMENT x RESULTAT
```

Autrement dit, le programme est l'exécution de la fonction P qui reçoit en paramètre un triplet formé d'une séquence d'équations, un environnement et un espace résolu, pour retourner un triplet du même type. Le programme s'arrête lorsque la valeur retournée est identique à la valeur introduite.

Cela s'écrit comme suit : μPser
avec μ : la fonction de point fixe
P : la fonction énoncée ci-dessus et exposée plus loin
s : la séquence d'équations — s∈SEQ[EQUATION]
e : l'environnement — e∈ENVIRONNEMENT
r : l'espace résolu — r∈RESULTAT

En sémantique de point fixe, on a l'habitude d'omettre les parenthèses ; Pser signifie lancement de l'exécution de P qui demandera successivement 3 paramètres, il les trouve ... derrière lui.

La sémantique de la fonction μ peut s'exposer en lambda calcul :
$\mu = \lambda G.[(\lambda g.Ggg)(\lambda g.Ggg)]$

On peut décrire l'exécution de μ par

```
μPser ::
        abc := ser ;
    boucle:
        wxy := Pabc ;
        if (wxy ≠ abc) then goto boucle
        else return wxy
```

Mais c'est une description opérationnelle qui risque de révolter.

On remarquera dans ce qui suit qu'il est fait un usage intensif des opérateurs surdéfinis ; on se permettra par exemple d'écrire p+"texte"+t où chaque + désigne un opérateur bien spécifique : spécifique au type de l'objet sur lequel il porte ; ce n'est qu'une question d'habitude tout comme les écritures μPerXg sans parenthèses où une lettre majuscule désigne une fonction, une minuscule une donnée (qui est une fonction sans paramètre !) et μ l'opérateur de μ-calcul. On a évité par contre l'usage de la λ-notation : "Per avec r tel que rx=Zab" est plus simple à lire.

Sémantique de Pser

P : SEQ[EQUATION] x ENVIRONNEMENT x RESULTAT ->
SEQ[EQUATION] x ENVIRONNEMENT x RESULTAT

Pser = ser_1 avec r_1 tels que er_1=μQser

Explications :

P reçoit s la séquence d'équations à résoudre ainsi que e un environnement et r un ensemble de fonctions partiellement résolues. P va tenter d'améliorer r à partir de s et e. Pour ce faire, il va 'exécuter' μQser.

Comme on le verra ci-dessous, Qser renvoie le même ser si s est vide, sinon renvoie s'er' avec s' étant s sans sa première équation et r' étant r complété de ce qui a pu être calculé dans l'équation de tête de s.

Comme P 'exécute' μQser, toutes les équations seront calculées et P recevra en retour s'er' avec s' vide et r' étant r complété de tous les calculs sur toutes les équations.

Enfin P 'retourne' s (la séquence originale), e (l'environnement original) et r_1 le RESULTAT issu de μQser. On verra plus loin qu'il est retourné la séquence originale pour pouvoir tenir compte des définitions croisées.

On voit maintenant que pour lancer tout le programme il suffit de faire : μPse0 en considérant que 0 désigne un RESULTAT vide.
μPse0 donnera ser, il suffira d'extraire r et de le consulter pour connaître la valeur calculée de toute variable du système d'équations s.

Certains préféreront le langage suivant : μPse0 définit la sémantique du programme, c'est-à-dire la table des triplets (identificateurs de variables d'équations, valeurs de domaine et valeurs de codomaine) comme le point fixe de la transformation P sur les valeurs se0.

Sémantique de Qser

Q : SEQ[EQUATION] x ENVIRONNEMENT x RESULTAT -->
SEQ[EQUATION] x ENVIRONNEMENT x RESULTAT

Q[<tete>:=<expression><suite>]er_1 = <suite>er_2
avec er_2 tels que er_2 = E<tête><expression>er_1
Q[.]er = [.]er

Quelques explications :

. l'analyse sémantique peut fournir soit une séquence vide, ce qu'on a symbolisé par [.] puisque syntaxiquement un système d'équations est terminé par '.', soit une séquence qui comporte une équation puis une séquence ; on a désigné ici l'équation par <tête>:=<expression> [4] et la suite par <suite>.

---

4 : Syntaxiquement, il faut écrire <tête>:=<expression> ; mais la sémantique ici exposée n'a pas être modifiée si la syntaxe est différente pourvu que les abstractions <tête> et <expression> soient fournies. Semblablement, on n'a pas à savoir comment <tete> et <expression> sont transmis. Rappelons enfin que := n'est indiqué que pour accompagner la lecture.

. si Q reçoit <tete>:=<expression> alors il retournera une séquence raccourcie (composée uniquement de <suite>) ainsi qu'un RESULTAT r modifié : insertion d'une nouvelle table ; en tout cas il y a modification et le programme devra se poursuivre : la fonction μ relancera Q.

. er_2 = E<tête><expression>er_1 signifie que la fonction E retourne un nouveau RESULTAT : r sera complété de la table pour <tête> de l'évaluation de <expression>.

. si Q ne reçoit rien du tout (un système d'équations vide symbolisé par [.]), toutes les équations ont été transformées en objet de RESULTAT. Mais comme les équations sont lues séquentiellement et qu'une équation peut comporter un 'appel' à une variable déclarée plus loin, la valeur de celle-ci a été estimée à $\perp$ et le résultat rangé dans la table des RESULTAT n'est qu'une approximation. Voilà pourquoi la fonction P calcule un point fixe de Qser et ne se contente pas d'un seul appel.

Sémantique de Etxer

E : TETE x EXPRESSION x ENVIRONNEMENT x RESULTAT
--> ENVIRONNEMENT x RESULTAT

<tête><expression>er_1 = er_2

Il y a quatre possibilités pour <tête> :
<tête> = <nom>
= <type><nom>
= <nom>(<paramètres>)
= <type><nom>(<paramètres>)

Mais on va considérer ici que les équations sont complètement typées : l'analyseur sémantique a bouclé son travail.

E[<type><nom>][<expression>]er
t n x

= e(r+(n,$\perp$,t,$\perp$,Xxer)) (1)

E[Fonct:(<type>-><type>):<nom>(<formels>)][expression]er
d c n p x

= er_1 avec r_1 tel que

si $d \in Nuple_n$ (2)
alors
$\forall$ $y \in d$ ($\forall$ $0<1\leq n$
soit n_i le nom du ième paramètre formel dans p
soit t_i le type du ième type dans d
soit y_i la ième valeur du nuple y
$r_2=r+(n_i,\perp,t_i,\perp,y_i,0)$) $\wedge$ (3)
$r_3=r_2+(n,d,c,y,Xxer_2)$ $\wedge$ (4)
$r_1=r_3\setminus\{(n_i,_,_,_,_,_) \mid 0<i\leq n\}$) (5)
sinon
soit n_0 le nom de l'unique paramètre dans p
$\forall$ $y \in d$ $r_2=r+(n_0,\perp,d,\perp,y)$ $\wedge$ $r_3=r_2+(n,d,c,y,Xxer_2)$ $\wedge$
$r_1=r_3\setminus\{(n_0,\perp,d,\perp,y)\}$

Note : les noms des paramètres formels ne peuvent exister dans r, mais c'est là un problème de l'analyseur sémantique.

Explications :

. (1) e(r+(n,⊥,t,⊥,Xxer)) est le doublet composé de l'environnement e et de l'espace de résultats r qui a été complété (avec conservation de la monotonie, confer plus haut la fonction + dans RESULTAT). Xxer sera défini ci-dessous ; c'est la fonction d'évaluation d'une expression qui retourne v∈VALEUR. On s'appuie sur l'analyseur sémantique pour certifier que v∈t (est du bon type). La valeur v doit être telle que si (n,⊥,t,⊥,x) existait déjà dans r alors $x \leq v$: une estimation fournit toujours un résultat au moins aussi bon.

. (2) si la variable est de type fonction, elle doit recevoir des paramètres, on suppose que l'analyseur sémantique a fait son travail et que si le domaine est du type $Nuple_n$, il y a n paramètres dont les types sont définis par d, sinon il n'y a qu'un paramètre de type d. Remarque : on verra plus loin que les uniques paramètres de type $Nuple_n$ ont été transformés en $Nuple_1$ dont l'élément est un $Nuple_n$.

. (3) $r_2=r+(n_i,\perp,t_i,\perp,y_i,0)$ est nécessaire pour que l'exécution du calcul de l'expression trouve dans son espace résolu la valeur des paramètres.

. (4) $r_3=r_2+(n,d,c,y,Xxer_2)$ ici on calcule la valeur de la fonction n(y). $Xxer_2$ trouvera dans r_2 toutes les variables paramètres avec leur valeur selon y.

. (5) $r_1=r_3\backslash\{(n_i,_,_,_,_,_) \mid 0<i\leq n\}$ enlève de l'espace résolu tous les paramètres ; ils n'ont plus rien à y faire, mais surtout une exécution pour une autre valeur de y demandera peut-être une valeur de paramètre 'plus petite' ce qui sera refusé par la règle d'insertion dans un espace résolu.

Sémantique de Xxer

X : EXPRESSION x ENVIRONNEMENT x RESULTAT -> VALEUR

Comme on est en notation polonaise postfixée, on va travailler avec une pile ; cette pile sera initialisée à vide et le tout transmis à Y comme suit : (0 désigne ici une pile vide)

$Xxer = v$ tel que $\mu Y\perp 0xer=vpx_1er$

On verra plus loin que ce ⊥ n'est pas utilisé (sera la dernière valeur calculée)

Sémantique de Yvpxer

Y : VALEUR x PILE x EXPRESSION x ENVIRONNEMENT x RESULTAT ->
VALEUR x PILE x EXPRESSION x ENVIRONNEMENT x RESULTAT

PILE n'a pas été défini, on supposera simplement qu'existent :

l'opérateur + tel que p+v est la pile p où v a été ajouté en tête

l'opérateur () tel que p(i) donne le ième élément de la pile avec p(0) désignant le premier élément

l'opérateur - tel que p-n est la pile p à qui l'on a ôté ses n premiers éléments

pile_vide(p) qui retourne vrai si la pile p est vide

Une analyse de ce qui suit montrera qu'en pile ne se trouvent jamais que des valeurs.

Remarque : on suppose comme d'habitude qu'aucune erreur d'exécution n'est possible.

```
Yvp[]er = vp[]er                                                  (1)

Yap[<nom><suite>]er =
     si ∃(<nom>,d,c,x,y)∈r                                        (2)
     alors si d=⊥
           alors y(p+y)[<suite>]er                                (3)
       sinon si d=Nuple_n
             alors soit z∈Nuple_n | ∀0<i≤n z_i=p(n-i)             (4)
                   soit (<nom>,d,c,z,y)∈r                         (5)
                   y((p-n)+y)[<suite>]er                          (6)
             sinon soit (<nom>,d,c,p(0),y)∈r
                   y((p-1)+y)[<suite>]er
     sinon si ∃(<nom>,d,c,x,y)∈e
           alors si d=Nuple_n
                 alors soit z∈Nuple_n | ∀0<i≤n z_i=p(n-i)
                       soit (<nom>,d,c,z,y)∈e
                       y((p-n)+y)[<suite>]er
                 sinon soit (<nom>,d,c,p(0),y)∈e
                       y((p-1)+y)[<suite>]er
           sinon ⊥0[]er                                           (7)

Yap[<valeur><suite>]er = <valeur>(p+<valeur>)[<suite>]er

Yap[if<type><expression>else<expression>endif<suite>]             (8)
        t       e_1                e_2

     = si a=⊥ ∨ a=⊤
       alors t_1p[<suite>]er                                      (9)
               avec si a=⊥ t_1=⊥ du type t
                    sinon t_1=⊤ du type t
       sinon si a=vrai
             alors si pile_vide(p-1)
                   alors ⊥(p-1)[e_1<suite>]er
                   sinon (p(1)(p-1)[e_1<suite>]er
             sinon si pile_vide(p-1)
                   alors ⊥(p-1)[e_2<suite>]er
                   sinon (p(1)(p-1)[e_2<suite>]er
Yap[<input><suite>]er = ⊥(p+⊥)[<suite>]er                        (10)
Yap[<output><parametre><suite>]er = [<suite>]er                   (10)
```

Explications :

. (1) cas de base

. (2) on a annoncé plus haut que les noms de fonctions de l'environnement ne sont plus surdéfinis au moment de l'exécution.
il ne peut donc y avoir de confusion entre nom.

. (3) variable sans paramètre, on saisit sa dernière approximation

. (4) z est la valeur de codomaine de la fonction

. (5) les tables sont totales, l'uplet existe forcément

. (6) ((p-n)+y) : p-n est la pile sans n éléments de tête, ensuite, le résultat y est empilé
il s'agit de $\bot_{Nul:}$ qui sera compatible avec tout type.

. (7) l'équation de cette variable n'a pas encore été traitée, sa valeur est donc estimée à $\bot$.

. (8) pour rappel "if a then b else c endif" donne "a if b else c endif" ; on se rappellera aussi que tout if est typé : son type est le type de la valeur à retourner en cas de test donnant $\bot$ ou $\top$

. (9) si le test donne $\bot$ toute l'expression ifthenelse retourne $\bot$, c'est l'implémentation monotone traditionnelle du if_then_else.

. (10) ce cas sera réglé ci-dessous : c'est le problème des effets de bord.

La sémantique ici exposée calcule toutes les tables de chaque équation (ou toute la table de la fonction "système d'équation"), on aurait tout aussi bien pu ne présenter que la sémantique d'un sous-but : le calcul d'une seule équation (qui en appellera probablement d'autres). Avantage : on peut donner un sens à **input** et **output**, mais l'énoncé se complique du fait qu'on ne peut exécuter les équations qui ne sont pas nécessaires au problème posé car il y aura des effets de bord déroutant au regard de la question posée.

La sémantique du calcul d'un sous-but peut être exposée en considérant une fonction de transformation de système d'équations qui ne retourne que la liste des équations strictement nécessaires. Il suffira alors d'appliquer la sémantique exposée plus haut à ce sous-ensemble (cohérent) d'équations.

Avant d'exposer cette sémantique, une remarque : un sous-but ne pourra, dans la réalité, être l'équation d'une variable-fonction : cela n'a pas de sens de demander la valeur de "factorielle", mais on se le permettra ici.

Soit donc la fonction

Z : SEQ[EQUATION] x SEQ[EQUATION] x SEQ[EQUATION] ->
    SEQ[EQUATION] x SEQ[EQUATION] x SEQ[EQUATION]

Zsbr = $s_1b_1r_1$ — s système d'équation à réduire
b une séquence de sous-buts qui ne se retrouvent par en r
r système d'équation réduit

la solution sera évidemment, (on commence à avoir le truc...)
μZsb0 avec b la séquence des buts à atteindre (supposée correct et sans doublons)
0 séquence vide

Zs[]r = s[]r
Zs[<but><suite>]r = sBs<suite>sr

B : SEQ[EQUATION] x SEQ[EQUATION] x SEQ[EQUATION] ->
    SEQ[EQUATION] x SEQ[EQUATION]

Bsubr reçoit
s le système d'équation original
u un sous-but
b liste de sous-buts restant à atteindre
r système d'équations réduit

retourne
b nouvelle liste de sous-buts
r système d'équations réduit complété

la liste de sous-buts restant à atteindre sera plus longue en cas de, par exemple but a avec a:=b+c+d

Bsubr = b_1r_1 avec b_1 et r_1 tels que
seb_1r_1=μEs<expression>br

on suppose ici que le but u qui est une équation est syntaxiquement décomposable en <tete>:=<expression>
il restera à la fonction E, que l'on ne va pas décrire à lire un à un les éléments de l'expression de telle sorte que, si elle rencontre un identifiacteur de variable d'équations elle effectue :

si l'équation de cette variable est dans r
alors rien
sinon rechercher son équation dans s
    ajouter cette équation aux sous-buts b.

Sémantique de l'analyse sémantique d'un système d'équations syntaxiquement correct

Notre programme d'exécution exposé ci-dessus nécessite un système d'équation "sémantiquement correct" ; autrement dit, si on lui fournit un système d'équation correct à son point de vue, il saura nous en calculer le point fixe. Et si le système d'équation est correct, il n'y a nul besoin d'analyseur sémantique. Evident ! pourtant, il y a là de quoi se tirer d'affaires.

Le but premier de ce travail est d'aider à l'analyse statique. Ne pourrait-on élaborer une analyse abstraite d'un système d'équations qui soit capable d'en déterminer les types ? Que si !

La sémantique de l'analyseur sémantique sera exposée en présentant simplement un traducteur de systèmes d'équations quelconques en systèmes d'équations sur les types utilisés. Il suffira alors de fournir le texte du système traduit à notre programme d'exécution puis d'insérer les types ainsi obtenus dans le système d'équations de départ.

Expliquons d'abord par un exemple et un dessin comment l'analyse de type peut être effectuée.

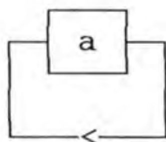
Soit le système d'équations quelque peu tordu :

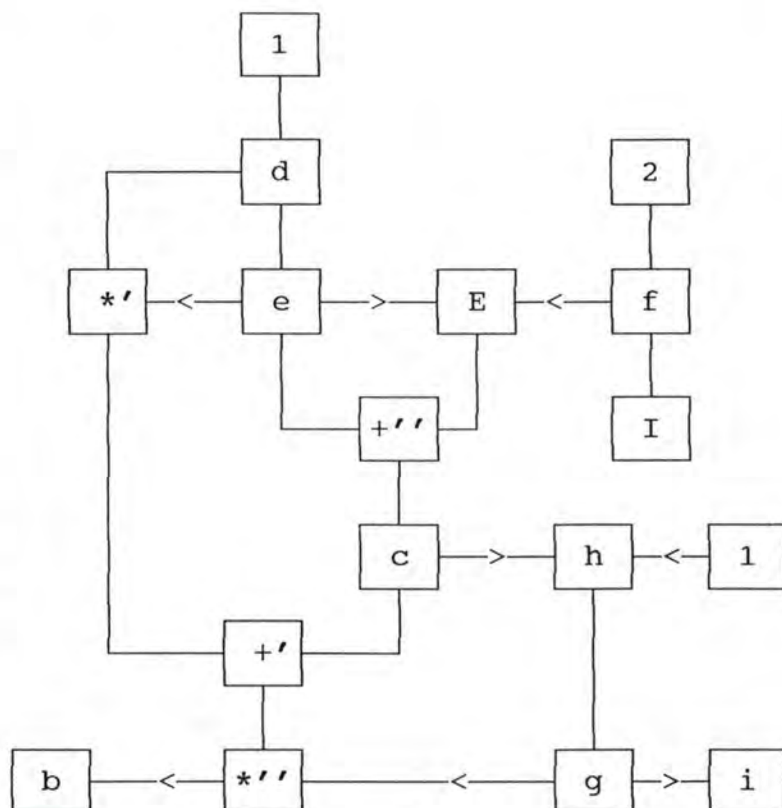
```
a := a ;
b := (c+d*e)*g(c) ;
c := e + if f then f else e endif ;
d := 1 ;
e := d ;
f := 2 ;
g(h) := h.
i := g(1) ;
```

Acceptons que les analyseurs lexicaux et syntaxiques aient vérifié tous les identificateurs et aient produit un système d'équations en notation polonaise postfixée ; ça donne

```
a := a
b := c d e * + c g *
c := e f IF f ELSE e ENDIF +
d := 1
e := d
f := 2
g(h) := h ;
i := g(1) ;
```

Dessinons le graphe des calculs :





les flèches non orientées descendent

E désigne ENDIF
I désigne IF

les deux '+' peuvent correspondre à des opérateurs probablement de types différents.

le type de b peut à coup sûr être déterminé en remontant le graphe ; ce qui n'est pas le cas de a.

Chaque flèche désigne une relation d'égalité de type entre domaines ou codomaine des objets qui constituent chaque élément de la séquence d'une équation en polonaise postfixée à savoir :

- codomaine d'une valeur
  une seule flèche qui en sort
- domaine de fonctions prédéfinies
  il y aura autant de fonctions à distinguer que d'appels de fonctions et donc autant de flèches qui y arrivent que de paramètres
- codomaine de fonction prédéfinies
  une seule flèche qui en sort
- codomaine de variable simples
  une seule occurrence par variable
  ce type sera défini par le type du codomaine de la dernière donnée dans l'équation
- domaine de variables-fonction
  une seule occurrence par nom de paramètre
  par nom de paramètre, autant de flèches qui en sort que d'appels à cette variable
- codomaine de variable-fonction
  autant de flèches qui en sortent que d'appels
- domaine d'un if : ce devra être un Pbool
- domaine d'un endif : deux flèches qui arrivent
- codomaine d'un endif : une seule flèche qui sort

Il est maintenant clair comment les types se déterminent les uns les autres ; on a les relations suivantes :

```
codom(a) = codom(a)
codom(b) = codom(*'')
codom(c) = codom(+'')+dom(g,1)
codom(d) = codom(1)+dom(*',1)
codom(e) = codom(d)+dom(*',2)+dom(E,2)
codom(f) = codom(2)+codom(I)+dom(E,1)
dom(+',1) = codom(*')
dom(+',2) = codom(c)
codom(+') = dom(*'',1)
dom(+'',1) = codom(e)
dom(+'',2)= codom(E)
codom(+'') = codom(c)
dom(*',1) = codom(d)
dom(*',2) = codom(e)
codom(*') = dom(+',1)
codom(I) = codom(f)
dom(E,1) = codom(f)
dom(E,2) = codom(e)
codom(E) = dom(+'',2)
dom(g,1) = codom(c)+codom(1)
codom(g) = dom(*'',2)+codom(i)
codom(i) = codom(g)
```

On ajoutera encore que le domaine de toute variable simple est $\perp$.
Ces relations peuvent être mises sous la forme d'un système d'équations et notre programme pourra en trouver le plus petit point fixe.
Malheureusement, comme on peut le constater ci-dessus, il faudra faire subir au texte une transformation radicale, car il ne serait pas suffisant de remplacer les opérateurs du lexique par des fonctions qui ne traitent que le type de la valeur rencontrée ; cette façon de procéder est intéressante mais ne permet de déterminer que les types des variables, pas le type des fonctions prédéfinies appelées si celles-ci sont surdéfinies, or c'est très souvent le cas, ni le type d'un if.

Soit donc un système d'équations composé de SEQ[EQUATION]

Avec chaque équation comportant l'une des 2 possibilités :

<type><nom>:=<expression>
<type><nom>(<paramètres>):=<expression>

On supposera que l'analyse syntaxique a fait en sorte que si le type d'une variable a été rencontré dans le corps d'une expression, ce type a été reporté au niveau de la <tête> exposée ci-dessus ainsi qu'à chaque occurrence d'appel et que chaque valeur est précédée de son type. Si aucun type n'a été rencontré, le type est Nul:.
L'analyse syntaxique a par ailleurs écarté les systèmes d'équations qui comportent plusieurs équations pour une même variable ou un appel à une fonction/variable inexistante.
De plus, l'analyse syntaxique fourni pour chaque fonction prédéfinie ou variable/fonction son arité (ceci grâce au système des parenthèses) et a donné comme type aux fonctions et variables-fonctions à plusieurs paramètres au moins le type Nuple_n avec n le nombre de paramètres (le minimum étant que tous les sous-types sont Nul:)

Avec <expression> étant une suite d'éléments parmi :

<type><valeur>
<type><nom_de_variable_simple>
<type><nom_de_variable_fct><noms_des_paramètres>
<type><fonction_prédéfinie><noms_des_paramètres>
<if>
<else>
<endif>
<input>
<output>

Supposons maintenant que chaque variable et chaque élément d'une équation puisse être identifié par un quelconque moyen ; notre tâche est d'attribuer un type 'maximal' à chacun de ces identifiants. Rappelons que chaque variable-paramètre d'une variable-fonction a un nom unique dans le système d'équation.
Chaque identifiant ci-dessus sera une variable du système d'équations à générer ; le type de chaque variable est TYPE décrit au tout début de ce chapitre (ainsi chaque variable est bien typée, et la sémantique d'exécution ci-dessus est d'application).

Un seul opérateur va nous être nécessaire :

+ : TYPE x TYPE -> TYPE : (a,b) -> c=lub(a,b)
ou a+b=c <=> c=lub(a,b)

Le symbole infixé + est plus commode que lub, on pourra écrire a+b+c (de même que dans un système d'équations avec <<=, mais + est plus court ..)

Supposons que l'on dispose d'opérateurs de concaténation de textes et d'une PILE comme exposée plus haut ; pile qui devra cette fois manipuler des textes.

L'analyse sémantique va peut-être rencontrer des erreurs, il y aura erreur quand la fonction lub ne peut donner de résultat par exemple lors de "Pint:a := Pbool:b". Pour gérer ces erreurs, on a introduit $\top$ dans le domaine des types. Ceci dit, on aurait pu considérer que toutes les fonctions sémantiques 'traînent' avec elle une valeur flag qui, lorsqu'elle est 'vrai', provoque l'abandon de tout calcul : on retourne les données comme on les a reçues sauf que f reste vrai.

On supposera que l'on peut écrire t=t+n+t+lub(x,y)+ident(e)

où t est un texte
n est un nom de variable
ident(e) l'identifiant d'un élément d'équation

ce n'est qu'un problème d'implémentation.

Allons-y :

Sémantique de la génération du texte

μTs0 avec s le système d'équation et 0 un texte vide

Explications :

. la fonction T, exposée ci-dessous, reçoit en paramètre une séquence à analyser s ainsi qu'un texte t et retourne une séquence raccourcie ainsi qu'un texte probablement complété

. l'initialisation est s=le système d'équation fourni par l'analyse syntaxique, t=0 un texte vide.

. le plus petit point fixe de la transformation T est calculé par μTs0.

Sémantique de Tst

T : SEQ[EQUATION] x TEXTE -> SEQ[EQUATION] x TEXTE

T[ ]t = [ ]t (1)

T[<equation>;<suite>]t = st$_1$ (2)
e s

avec t$_1$ tel que Wet = t$_1$ (3)

Explications :
. (1) point fixe atteint
. (2) si (1) n'est pas vrai alors la séquence peut être décomposée en [<equation>;<suite>] avec
. ';' inséré ici juste pour aider à la compréhension
. e désigne la première équation
. s désigne les équations restantes.
. (3) W est la fonction qui va compléter t de l'évaluation de l'équation e et retourner le nouveau texte.

Sémantique de Wet

W : EQUATION x TEXTE -> TEXTE (1)

W[<type><nom>:=<expression>]t = t$_1$ (2)
i n e

avec t$_1$ tel que
e$_2$pt$_2$=μVe0t $\wedge$ (3)
t1=t+"codom"+n+":="+i+"+"+p(0))+";" (4)
+"dom"+n+":=Nul:;"

W[<type><nom>(<paramètres>):=<expression>]t = t$_1$ (5)
i n p e

soit m le nombre de paramètres dans p
si m=1
alors t$_2$=t$_1$+"dom"+p+":=Nul:;codom"+p+":="+y (6)
avec y le type du domaine selon i
sinon mipt$_2$=μNmipt$_1$ $\wedge$ (7)
e$_2$qt$_2$=μVe0t $\wedge$ (3)
soit x le type du codomaine selon i
t1=t+"codom"+n+":="+x+"+"+q(0))+";" (8)

Explications :
. (1) il y a deux possibilités syntaxiques pour <équation> :
[<type><nom>:=<expression>] et
[<type><nom>(<paramètres>):=<expression>]
Il y a toujours un type minimal attribué par l'analyse syntaxique, et les symboles :=() ne sont indiqués que par commodité.
. (2) il faut vérifier si le type de l'expression e est compatible avec le type <type>
. (3) calcul du point fixe de Ve0t, (on verra plus loin que 0 est une pile initialisée à vide) ce qui retournera e$_2$ une expression vide avec en tête de pile p ce qui permettra d'identifier le type de l'expression e (le codom d'une variable par exemple ou le codom d'un if)
. (4) rappelons que lub est total ; Nul: est le cas de base ; "codom"+n : ce qui identifie le codom de la variable

i : le type déclaré syntaxiquement
p(0) : ce qui permettra d'identifier le type de l'expression
Il n'est pas nécessaire de signaler que le domaine d'une variable paramètre est Nul:.

. (5) n est une variable fonction, chaque paramètre a un nom unique dans le système. Si l'arité est 1, le type de l'unique paramètre est i ; si l'arité est >1, i est du type $Nuple_n$ avec n l'arité.

. (6) on insère le type de l'unique paramètre dont le nom est p en disant que son domaine est Nul: et son codomaine p (on suppose qu'il n'y a pas de spécification de type dans la liste des identificateurs de paramètres, ce serait redondant avec i).

. (7) la fonction N exposée ci-dessous insère dans le texte la définition du type (trouvé dans i) de chaque paramètre au regard de son nom (trouvé dans p) supposé unique ; m servira de compteur : quand m est à zéro, le point fixe est atteint.

. (8) (simple renvoi pour une remarque qui sera faite plus tard)

Sémantique de Nmipt

Nm<type><paramètres>t = Nm_1ipt_1
i p

N0ipt = 0ipt
Nmipt = (m-1)ipt_1 avec t_1 tel que
soit t_i le type du ième paramètre selon i
soit n_i le nom du ième paramètre selon p
t_1=t+"dom"+n_i+":="t_i+";"

Sémantique de Vept (1)

Vept = $Ve_1p_1t_1$ tel que

V[]pt = []pt (2)

V[<type><nom_de_variable_simple><suite>]pt =
i n s

s(p+("codom"+i))t (3)

V[<type><valeur><suite>]pt =
i v s

s(p+i)t (4)

V[<type><var-fonction><noms_paramètres><suite>]pt =
i n r s

sp_1t_1 avec p_1 et t_1 définis par : (5)

soit a l'arité de r
$a_2rp_2t_1=\mu Marpt$ (6)
soit j le type du codomaine de i
$p_1=p_2$+("codom"+n) (7)

V[<type><fonction><noms_paramètres><suite>]pt =
i n r s

sp_1t_1 avec p_1 et t_1 définis par : (8)

soit k l'identifiant de cet appel
soit a l'arité de r

$a_2krp_2t_1=\mu Fakrpt$ (9)
soit j le type du codomaine de i
$p_1=p_2+j$ (10)

V[<if><suite><else><suite><endif><suite>]pt = (11)
s_1 s_2 s_3

$s_3p_1t_1$ avec p_1 et t_1 définis par : (12)

soit i l'identifiant de la localisation de if
$t_2=t+\text{"dom"}+i+\text{":="}+p(0) \wedge$ (13)
$xp_3t_3=\mu Vs_1 0t_2 \wedge$ (14)
$yp_4t_4=\mu Vs_2 0t_3 \wedge$ (15)
$t_5=t_4+\text{"codom"}+i+\text{":="}+p_3(0)+\text{"+"}+p_4(0)+\text{";"} \wedge$ (16)
$t_1=t_5+p_3(0)+\text{":="}+p_4(0) \wedge$ (17)
$p_1=(p-1)+p_4(0)$ (18)

Explications :

. (1) Cette fonction Vept reçoit une pile p avec des textes qui identifient un "lieu" dans une expression ou une variable ; e est l'expression à analyser et t le texte où inscrire les relations de types rencontrées lors de l'analyse de e. Les cas suivants sont à analyser :

[]
[<type><nom_de_variable_simple><suite>]
[<type><valeur><suite>]
[<type><var-fonction><noms_paramètres><suite>]
[<type><fonction><suite>]
[<if><suite><else><suite><endif><suite>]
[<input>] [<output>]

. (2) le point fixe est atteint

. (3) on a rencontré une variable, cette variable (ou plutôt sa valeur) sera utilisée comme paramètre d'une fonction ou comme résultat d'une équation ; il faut ranger son type en tête de pile par p+i (qui a défini plus haut) ; en fait, il faut ranger l'identifiant du type de son codom, à savoir "codom"+n si n est son nom.
Dans l'écriture p+("codom"+i) on n'a peur de mélanger les genres : le premier '+' est une ajoute en pile, le second est une concaténation d'une chaîne constante avec la chaîne désignée par une variable.

. (4) la valeur n'a pas d'intérêt, seul son type est enregistré dans la pile ; ce sera un cas de base.

. (5) cas d'un appel de variable de type fonction ; lors de l'exécution concrète, cette 'fonction' devra aller rechercher ses paramètres dans la pile et ranger son résultat dans la pile ; en exécution abstraite, il faut associer à chaque codomaine de paramètre formel ce qu'il y a dans la pile. Chaque paramètre formel est une variable identifiée par un nom unique (si nécessaire : concaténation du nom de la variable et du paramètre formel).

. (6) Marpt va insérer dans le texte les types des paramètres effectifs trouvés dans la pile et les associer aux paramètres formels.

. (7) le codomaine de la variable doit être inséré en pile ; car le résultat de l'appel de la variable-fonction va devenir un argument de fonction ou le résultat de l'équation. Remarquons qu'on n'insère pas dans la pile le codomaine du type déclaré i mais "codom"+n car lors de la lecture de l'équation de n on insérera "codom"+n":="+ (le codom trouvé en i) ; en effet il se peut que les deux ne correspondent pas. Confer plus haut la remarque (8) dans la définition de **Wet**.

. (8) cas d'un appel de fonction ; cette fonction doit aller rechercher ses paramètres dans la pile et ranger son résultat dans la pile. Il est indispensable de connaître l'arité de la fonction. Le nom de la fonction ne suffit pas à son identification ; en effet une même fonction virtuelle peut exister en divers endroits du système d'équa-

tions avec des paramètres de types distintcs : ce seront des fonctions différentes (on a supposé plus haut disposer d'un identifiant pour chaque appel de fonction) ; cette remarque n'est pas d'application aux variables-fonctions. Le type du codomaine de la fonction ne peut aider à déterminer la fonction, mais il doit être vérifié.

. (9) confer (6) mais l'identifiant sera ici le nom du paramètre + la localisation de l'appel.

. (10) à la différence de (7), on doit ici mettre le type déclaré en pile.

. (11) ifthenelse est considéré comme une fonction à trois paramètres, son premier paramètre doit être du type booléen, ses deux suivants d'un même type qui sera celui du codomaine de la fonction ifthenelse. On devra exécuter abstraitement les deux sous-expressions ; l'écriture suivante étant interdite

a:= if b then c , d , e else d endif ;

on peut accepter que l'exécution de l'expression du then ne mettra qu'une seule donnée en pile.

. (12) si i est l'identifiant du if, "dom"+i devra être Pbool:, "codom"+i sera le type de la valeur retournée, on aura donc deux écritures "codom+i := ..." l'une pour le then, l'autre pour le "else" ; ensuite on aura ".... := codom+i", ce sera l'utilisation de la valeur de l'expression ifthenelse.

. (13) insertion en texte du type qui est en pile (et qui devra être Pbool:) associé au dom du if

. (14) exécution abstraite de l'expression entre then et else

. (15) exécution abstraite de l'expression entre else et endif

. (16) insertion en texte que le codom du if est le lub du type de l'expression entre then et else et l'expression entre else et endif

. (17) insertion en texte que ces deux expressions doivent être du même type

. (18) enfin on poursuit en mettant en pile l'un des deux types d'expression, au choix.

Sémantique de Marpt

Marpt = $a_1 r p_1 t_1$ tel que :

M0rpt = 0rpt

Marpt = (a-1)r(p-1)t_1 tel que

soit j le nom du paramètre numéro a dans r

t_1=t+"codom"+j+":="+p(0)

PRINCIPAUX ALGORITHMES

ALGORITHME DE GENERATION D'UNE EQUATION A PARTIR D'UN TEXTE

Ce qui est présenté ci-dessous est une simplification.

```
équation ::
lire le nom de la variable
lire ':='
initialiser pile principale à vide
initialiser pile secondaire à vide
exécuter expression
lire le symbole S
si S est ";" reprendre
si S n'est pas '.' erreur

expression ::
lire le symbole S
si S est '('
alors mettre '(' en pile secondaire
      exécuter expression
sinon si S est <fonction>
      alors lire le symbole suivant qui doit être '('
         exécuter expression
      sinon si S n'est pas <var> ou <cste> alors erreur
      mettre S en pile
si pas fin du texte
alors lire le symbole S
      si S n'est pas ')'
      alors si S pas opérateur erreur
         soit n le niveau de S
         recopier de pile secondaire tous les opérateurs plus petits
         mettre S en pile secondaire
         aller en expression
recopier de pile secondaire tous les opérateurs
  jusqu'à fin ou '('
```

ALGORITHME D'ANALYSE DE TYPE

Plus précisément : algorithme d'analyse de type d'un système d'équations en notation polonaise postfixée.

0) rappel des données disponibles

L'analyse sémantique est faite sur un système d'équations ; c'est-à-dire :

0. un lexique d'opérateurs et fonctions prédéfinies
classés selon
   - le nom (non identifiant !!!)
   - l'arité

   s'il existe plusieurs fonctions de même nom et arité
   il existe d'abord une fonction dont le type est
   Fct:(0-->0):
   qui est capable de lancer l'exécution d'une des
   fonctions qui suit selon le type des paramètres
1. une liste de n variables
chaque variable étant caractérisée par
   - un nom identifiant
   - une arité
   - un type (ou nul ou 0 ...)
2. une liste de n suites d'équations
chaque élément d'équation est l'un des
   - une valeur (type disponible immédiatement)
   - un identificateur de variable
   - un identificateur de fonction
   - IF ou ELSE ou ENDIF
     accompagné du type de valeur retournée par
     l'expression qui précède.

1) génération du système des équations de type

Ce système se composera de lignes du type suivant :
<nom> := <texte>
<nom> est un texte qui va identifier l'un des :
domaine du ième paramètre d'opérateur ou fonction
domaine de ième paramètre d'une variable
domaine d'un if
codomaine de variable
codomaine d'opérateur ou fonction
codomaine d'un if
<texte> est défini par <texte> :: <nom> | <type> | <texte> + <texte>
<type> est une constante : un objet caractérisant un type.
+ est la fonction qui calcule le lub de deux types ou types de variable.

Concrètement un nom sera composé comme suit :
domaine du ième paramètre d'opérateur ou fonction :
"dom"+adresse+i
domaine de ième paramètre d'une variable :
"dom"+adresse+i
domaine d'un if
"dom"+adresse
codomaine de variable :
"codom"+adresse

```
codomaine d'opérateur ou fonction :
    "codom"+adresse
codomaine d'un if
    "codom"+adresse
```

On voit qu'on obtiendra un système d'équations où toutes les variables sont du type Typ: et dont le point fixe sera la meilleure approximation possible des types des variables, opérateurs et fonctions utilisées dans le système analysé.

Pour générer ce texte, on va utiliser une structure de données intermédiaire :

T est une table de couples (<nom>,<texte>)
    avec <nom> et <texte> comme exposé plus haut.

T est muni de l'opérateur infixe <- défini par

```
T2 = T1<-(a,b)
<=>
si a n'existe pas dans T comme premier élément d'un couple
alors T2 est T1 complété de l'élément (a,b)
sinon soit (a,x) appartenant à T1
      T2 est T1 sans (a,x) et complété de (a,y)
      tel que y est la concaténation de x '+' b
```

Voici enfin l'algorithme :

Initialiser T à vide

Pour tout x variable d'équation effectuer :

```
   T <- ('codom'+ adresse_de(x),type_du_codom_déclaré_de(x))
   si l'arité de x est zéro
   alors  T <- ('dom' + adresse_de(x) + '0',Nul)
   sinon Pour i de 1 à l'arité de x
              T <- ('dom' + adresse_de(x) + i ,
                      type_déclaré_du_ième_paramètre)
   Soit e l'adresse de l'équation de x
   Tant que e<>0 effectuer :
        soit ee=e
     si *e est un opérateur ou une fonction ou IF ou ELSE ou ENDIF
        alors Pour i de 1 à l'arité de cet opérateur ou fonction
              effectuer :
                 e=ee ; remonter de n-i+1
                 si e désigne une valeur
                 alors T <- ('dom' + ee + i , type_de_la_valeur)
                 sinonsi e désigne une variable , disons y
                 alors T <- ('dom' + ee + i , 'codom' + adresse_de(y))
                       T <- ('codom' + adresse_de(y) , 'dom' + ee +i)
                 sinonsi e désigne un opérateur ou fonction ou ENDIF
                 alors T <- ('dom' + ee + i , 'codom' + ee)
                       T <- ('codom' + e , 'dom' + ee + i)
        sinonsi *e est une variable, disons y
                si l'arité de *e est 0
                alors rien
                sinon Pour i de 1 à l'arité de cette variable
                      effectuer :
                         e=ee ; remonter de n-i+1
                         si e désigne une valeur
                         alors T <- ('dom'+ adresse_de(y) + i,
```

```
                                          type_de_la_valeur)
                              sinonsi e désigne une variable
                              alors T <- ('dom'+ adresse_de(y) + i,'
                                          codom'+ adresse_de(x))
                                    T <- ('codom' + adresse_de(x) ,
                                          'dom' + ee + i)
                              sinonsi e désigne un opérateur ou fonction
                                                ou endif
                              alors T <- ('dom' + adresse_de(y) + i,
                                          'codom'+ ee)
                                    T <- ('codom' + e,
                                          'dom' + adresse_de(y) + i)
             sinon on a une valeur et on ne fait rien
             e++ ;
        remonter de 1
        si e désigne une valeur
        alors T <- ('dom' + adresse_de_x , type_de_la_valeur)
        sinonsi e désigne une variable , disons y
        alors T <- ('dom' + adresse_de_x , 'codom' + adresse_de(y))
              T <- ('codom' + adresse_de(y) , 'dom' + adresse_de_(y))
        sinon e désigne un opérateur ou fonction ou ENDIF
              T <- ('dom' + adresse_de(x) , 'codom' + e)
              T <- ('codom' + e , 'dom' + adresse_de(x))
```

ALGORITHME D'EXECUTION D'UNE PILE D'EQUATIONS

La notion de pile a déjà été définie dans l'exposé sémantique du programme.

```
   recevoir une pile principale et une pile secondaire
boucle ::
   si pile principale vide
        sortir (le résultat est en pile secondaire)
   soit e la tête de la pile principale
   extraire e, tête de la pile principale
   si e est une data
        mettre e en tête de pile secondaire
        aller en boucle
   si e est ENDIF
        aller en boucle
   se i est ELSE
        exécuter n=rechercher(ENDIF)
        extraire n équations de la pile principale
        aller en répéter
   si e est IF
        (erreur si pile secondaire vide)
        soit b la tête de la pile secondaire
        extraire b, tête de la pile secondaire
        (erreur si b n'est pas du type boolen)
        si b est BOTTOM
            exécuter n=rechercher(ENDIF)
            extraire n équations de la pile principale
            mettre en pile secondaire la valeur bottom de e
            aller en boucle
        si b est VRAI
            aller en boucle
        si b est FAUX
            exécuter n=rechercher(ELSE)
            extraire n équations de la pile principale
            aller en répéter
   Exécuter(e)
   aller en répéter

Algorithme de rechercher(x)

encore :
   (erreur si pile principale vide)
   soit e la tête de la pile principale
   extraire e de la pile principale
   si e est x
     sortir
   si e est IF
     rechercher(ENDIF)
   aller en encore
```

"Exécuter(e)" signifie qu'il a été rencontré soit une variable soit une variable-fonction soit une fonction prédéfinie ; dans tous les cas, le programme fait un appel au gestionnaire d'appels récursifs qui :

- pour une variable simple : appel de tau (voir ci-dessous)
- pour une variable-fonction : va chercher les valeurs des paramètres effectifs dans la pile, puis appel de tau
- pour une fonction prédéfinie : c'est la fonction qui doit aller rechercher les paramètres effectifs dans la pile.

Dans chacun de ces cas une valeur est retournée et est rangée dans la pile.

Le gestionnaire d'appels récursifs a - entre autres - pour tâche d'adapter les types des valeurs manipulées ; en effet, l'exécuteur d'équation ainsi que les fonctions prédéfines manipulent des données du type déclaré dans le programme, alors que le processeur de point fixe considère tout le système d'équations comme une seule fonction.

La fonction "système d'équation" a pour type :

Fct:(Couple:(Pint:,Union:(D_1,D_2,...D_n):):):->
Union:(C_1,C_2,...C_n):):

avec D_i le domaine de la variable d'équation i
C_i le codomaine de la variable d'équation i.

Signalons par ailleurs, que la version actuelle du programme est très redondante. Cette redondance a été instaurée en vue de faciliter un suivi du déroulement et d'éventuelles correction. Avantage : chaque équation traîne avec elle le "lexème" (unité syntaxique) qui l'a produite et l'on peut demander à voir l'éxécution pas-à-pas de son système d'équations.

Trois algorithmes distincts de point fixe sont livrés dans le programme, ils sont attachés aux objets de classe de "fonction" ; le solveur de systèmes d'équations n'exploite que le dernier.

ALGORITHME DE POINT FIXE NUMERO 0

simple : on recalcule tant qu'on parvient à améliorer ;
tout calcul sur une donnée en cours est abandonné :
on utilise sa dernière approximation ;
toute insertion en table respecte la monotonie.

ALGORITHME DE POINT FIXE NUMERO 1

on effectue un calcul de f(a) uniquement
soit si a n'est pas en table
auquel cas il l'insère d'abord comme instable
avec comme valeur le lub des plus petits que lui
soit si a est 'instable' et 'pasencours'

si l'on effectue le calcul de a :
pendant ce calcul : a est déclaré 'encours'
après ce calcul : a est déclaré 'pasencours'

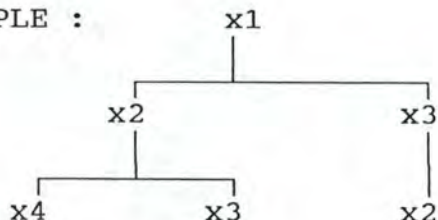
on déclare a 'stable'
si toute les données utilisées pour son calcul sont stables
on déclare a 'instable'
soit si le calcul de a est déjà en cours
soit si une des données nécessaires est instable

On obtient un point fixe en
répétant ce qui est exposé ci-dessus
jusqu'à ce que stable
ou que le recalcul n'améliore rien

L'algorithme est présenté à la page suivante, on notera qu'il est demandé que tout appel récursif de calcul soit effectué en direction de "tau" par l'intermédiaire d'un programme qu'on appelle ici "recursion" et que la fonction de calcul est priée de véhiculer un paramètre : un drapeau permettant de détecter la stabilité d'un calcul.

```
Soit un élément en table désigné par (a,b,s,c)
        avec    b=f(a)
                s : vrai si b est stable
                c : vrai si a en cours

 mu(a) ::
    initialiser table des valeurs à vide
    t=tau(a)
    si pas t
    alors répéter : soit b tel que (a,b,-,-) en table
                    t=tau(a)
                    soit b' tel que (a,b',-,-) en  table
          tant que pas t ni b=b'
    retourner b tel que (a,b,-,-) en table

 tau(a) ::
    si il existe (a,b,s,c) en table
                // s=vrai signifiant que b=f(a) : la valeur est stable
                // c=vrai signifiant que le calcul est en cours
    alors si c=vrai retourner s
                // sortie immédiate si le calcul de f(a) est en cours
          si s=vrai retourner s
                // sortie immédiate si f(a) est totalement déterminé
    sinon b = lub{ b'|(a',b',-,-) en table et a'<=a }
          table = table + {(a,b,-,-)}
                // la table est initialisée avec la meilleure
                // approximation initiale b=f(a) suffisante
                // pour garantir la monotonie de la table
    table = table - {(a,b,-,-)} + {(a,b,faux,vrai)}
                // s=faux : a est instable
                // c=vrai : le calcul de f(a) est en cours
    s = vrai
    y = calcul(a,s&)
                // en sortie de calcul, s sera vrai si tous les
                // calculs ont donné des valeurs stables
                // si s est vrai y=f(a) est stable
    table = table - {(a,b,-,-)} + {(a,b,s,faux)}
                // la calcul de f(a) n'est plus en cours
    table = table - {(a',-,-) | a<=a'}
            + {(a',y,s,c) | (a',x,s,c) en table et a<=a'
                            et y=lub(z) tel que (a'',z,-,-) en table
                            et a''<=a'
                // nécessaire pour garantir la monotonie de la table
    retourner s

  calcul(a,s) ::
     .....
                // l'appel récursif y=fonction(x) est remplacé par
                // l'appel à la fonction de récursion qui est :

   recursion(a,s) ::
      s = s && tau(x)
      y tel que (x,y,-,-,) en table
                // l'appel tau(x) fournit une approximation
                // elle sera lubmale si s=vrai
   .....
```

EXEMPLE :

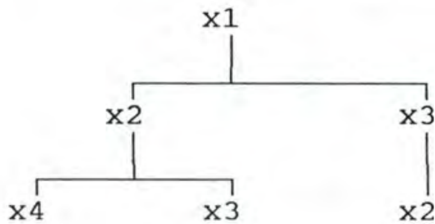


x1 est fonction de x2 et de x3
x2 est fonction de x3 et de x4
x3 est fonction de x2
x4 n'est fonction de personne (calculable immédiatement)

```
tau(x1) :
  x1 est inconnu, on insère x1 en table avec
         x1 -> (BOTTOM,passtable,encours)
  le calcul de x1 demande le calcul de x2
  tau(x2)
    x2 est inconnu, on insère x2 en table avec
           x2 -> (BOTTOM,passtable,encours)
    le calcul de x2 demande le calcul de x4
    tau(x4)
      x4 est inconnu, on insère x4 en table avec
             x4 -> (BOTTOM,passtable,encours)
      le calcul de x4 donne un résultat définitif
             x4 -> (brol,stable,pasencours)
    le calcul de x2 demande le calcul de x3
    tau(x3)
      x3 est inconnu, on insère x3 en table avec
             x3 -> (BOTTOM,passtable,encours)
      le calcul de x3 demande le calcul de x2
      tau(x2)
        x2 est en cours, faut se contenter de son approximation
      le calcul de x3 est terminé
                 x3 -> (brol,passtable,pasencours)
    le calcul de x2 est terminé
               x2 -> (brol,passtable,pasencours)
  le calcul de x1 demande le calcul de x3
  tau(x3)
    x3 est en table mais n'est pas stable
       le calcul de x3 demande le calcul de x2
    tau(x2)
      x2 est en table mais n'est pas stable
         le  calcul de  x2 demande le calcul de x3
      tau(x3)
        x3 est en cours, faut se contenter de son approximation
      le calcul de x2 est terminé
                 x2 -> (brol,passtable,pasencours)
  le calcul de x1 est terminé
           x1 -> (brol,passtable,pasencours)
le calcul de x1 n'a pas donné un résultat stable, faut reprendre
mais on mémorise la valeur déjà obtenue
tau(x1) :
 ..... idem que ci-dessus
on recommence tant que ça ne s'améliore pas
```

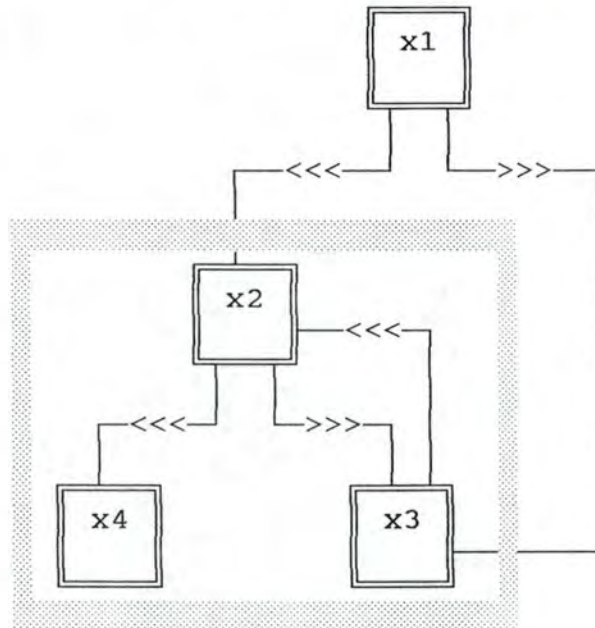
ALGORITHME DE POINT FIXE NUMERO 2

Si l'on reprend l'exemple plus haut :



On constate que l'algorithme numéro 1 effectue au moins deux fois le calcul de x1. En effet le calcul de x3 se révèle instable puisque x2 est alors en cours.
Or f(x2) pourrait être complètement déterminé par tau(x2) : l'appel de tau(x2) dispose de toutes les données suffisantes pour rendre une valeur stable.
De la sorte, tau(x3) sera également immédiat et tau(x1) obtient une valeur stable dès le premier essai.

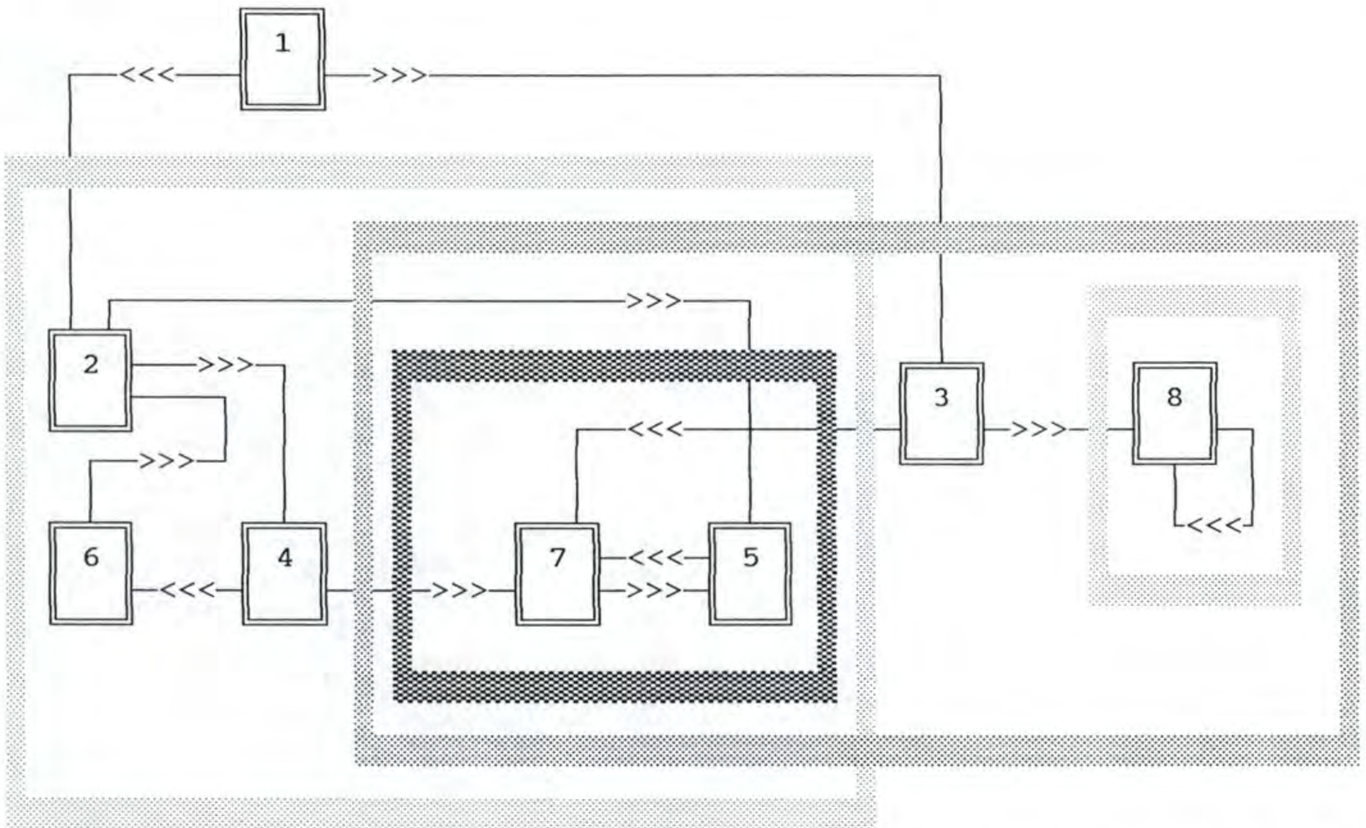
Si l'on dessine comme ci-dessous un "graphe des dépendances", on constate qu'aucune flèche ne sort de l'encadré. Il faut en conclure que la valeur de x2 doit pouvoir y être déterminée totalement.



Autre exemple :

x1 dépend de x2 et x3
x2 dépend de x4 et x5
x3 dépend de x7 et x8
x4 dépend de x6 et x7
x5 dépend de x7
x6 dépend de x2
x7 dépend de x5
x8 dépend de x8

On peut construire comme suit 4 rectangles d'où ne sort aucune flèche :



Au vu de ce dessin, la stratégie de calcul intuitivement optimale sera d'effectuer dans l'ordre :

	circuit :
calcul de x8	x8-x8
calcul de x5 et x7	x5-x7-x5
calcul de x3	
calcul de x2, x4 et x6	x2-x4-x6
calcul de x1	

Ces ordres de calculs sont en fait déterminés par la présence des circuits suivants :

x8 -> x8
x5 -> x7 -> x5
x2 -> x4 -> x6 -> x2

Ces deux premiers circuits se suffisent à eux-mêmes : aucune flèche ne sort de leurs sommets ; par contre le dernier nécessite de connaître x7 et x5 ; on dira que ce dernier n'est pas 'autonome'.
Remarquons que la présence d'un circuit même 'autonome' ne signifie pas qu'il soit utile de le calculer : le calcul de x8 est inutile à x2.
Bref, l'algorithme précédent peut être amélioré si l'on détecte la présence de circuits autonomes lors de la "descente des tau".

Notre exemple, ne devrait comporter que les appels à tau suivants :

```
tau(1) qui effectue
UNE SEULE FOIS :
    tau(2) qui effectue
    PLUSIEURS FOIS :
        tau(4) qui effectue
        PLUSIEURS FOIS :
```

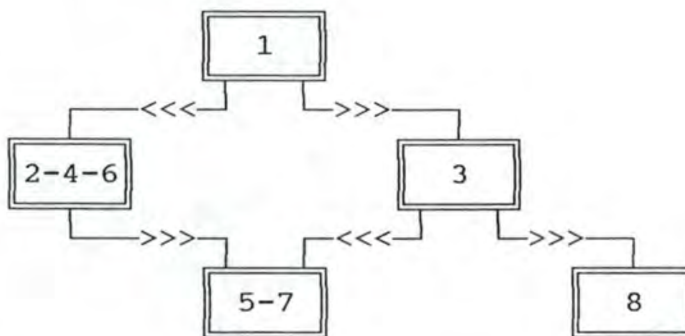
```
                tau(6)
            et UNE SEULE FOIS
                tau(7) qui effectue
                PLUSIEURS FOIS :
                    tau(5)
    et UNE SEULE FOIS
        tau(3) qui effectue
        UNE SEULE FOIS
            tau(7)
        et UNE SEULE FOIS
            tau(8) qui effectue PLUSIEURS FOIS son propre calcul
```

"une seule fois" quand on rentre dans un rectangle ;
"pusieurs fois" quand on reste dans son rectangle.

Les tau suivants obtiennent une valeur STABLE dès leur premier appel :
tau(1) tau(2) tau(3) tau(7) tau(8)
Les tau suivants vont rechercher un POINT FIXE secondaire :
tau(2) tau(7) tau(8)

Plus simplement : si l'on remplace dans le graphe des dépendances tout cycle par un seul sommet, on obtient un graphe qui indique clairement que le tau de chaque sommet obtiendra un résultat stable :

nouveaux sommets : 2-4-6 5-7 8



Notre algorithme va devoir faire en sorte qu'à la fin de chaque tau il puisse être déterminé s'il est nécessaire de calculer un point fixe secondaire.
Tout sommet peut déterminer s'il est nécessaire de calculer un point fixe secondaire en comparant la liste des calculs interrompus avec la liste des calculs antécédents : si un calcul interrompu est un antécédent, il faut renvoyer un résultat approximatif. Le sommet 6 ayant été appelé par 4 qui a été appelé par 2 doit suspendre son propre calcul ; par contre, le sommet 2 recevra une liste d'interrompus (4,6) qui ne comprend aucun de ses antécédents et doit donc réexécuter son calcul

Soit donc deux structures en pile :
. l'une reçue de son antécédent : la liste des ancêtres
. l'autre initialisée à vide avant un calcul
comportera après le calcul la liste des interrompus.
L'algorithme devient :

```
tau(a) ::
  si il existe (a,b,s,c) en table
  alors si s=vrai retourner vrai
        si c=vrai alors insérer son nom dans la liste des
                        interrompus et retourner faux
  sinon b = lub{ b'|(a',b',-,-) en table et a'<=a }
        table = table + {(a,b,-,-)}
  insérer a dans la liste des ancêtres
  boucle :
  créer une nouvelle liste d'interrompus
                initialisée à vide
  s = vrai
  y = calcul(a,s&)
  si s=vrai ou si dans la liste des interrompus ne se trouve que a
     et que b=y
  alors y est stable
        restaurer la liste des interrompus
        retirer a de la liste des ancêtres
        mettre à jour la table par
                table = table - {(a,b,-,-)} + {(a,y,vrai,faux)}
                table = table - {(a',-,-) | a<=a'}
                        + {(a',b',s,c) | (a',x,s,c) en table
                                         et a<=a' et b'=lub(x,y)
        retourner vrai
  retirer a de la liste des ancêtres
  si aucun nom de la liste des interrompus n'apparaît dans
        la liste des ancêtres
  alors supprimer la liste des interrompus
        et aller en boucle
  sinon ajouter tous les noms de la liste des interrompus à
           l'ancienne liste
        et y ajouter son nom
        supprimer la liste des interrompus
        restaurer l'ancienne liste
        retourner faux
```

Extensions

L'analyse des types (qu'on a appelée 'sémantique') nous a montré combien est ardue la traduction d'un programme en système d'équations en vue de son analyse statique ; de plus, les types fournis ne sont que rarement adéquats. Il serait bien utile de disposer d'un générateur d'équations ; on lui fournirait d'une part un texte, d'autre part une règle comportementale. Malheureusement, il semble bien compliqué de concevoir un générateur ouvert à tout type de programmes ; de plus, il faut que l'effort en vaille la peine ; il faut être certain que l'exécution du programme aura de meilleures performances.

Le logiciel ici mis au point s'il reste 'expérimental' mériterait tout de même quelques extensions dans les domaines où l'analyse statique rencontre le plus d'opportunités d'optimisation. On pense surtout au langage de programmation Prolog et il semble que l'ajout de domaines et opérateurs spécifiques soit aisé.

Tout au début de ce travail, il a été exposé que la forme "système d'équations" est somme toute assez universelle ; ce qui l'est plus encore, c'est la théorie du point fixe et sa version algorithmique, ils ont un très grand pouvoir de représentation (sémantique dénotationnelle) et de résolution. Une belle extension serait l'interface capable de recevoir les règles d'un langage (c'est un système d'équations !) puis capable d'en accepter des textes (analyse par la syntaxe comme nous avons fait dans le chapitre applications).

Mais avant de penser aux extensions, il faudrait d'abord :

. terminer les codes de l'analyseur "sémantique" ;

. améliorer l'interface utilisateur qui est très rudimentaire ;

. optimiser l'exécution d'une équation (une gestion de mémoire a été instaurée lors des analyses lexicales et sémantiques ainsi qu'une optimisation des accès ; l'effort n'a pas encore été suffisamment appliqué au module d'exécution)

. profiter de la "descente" des équations lors de l'analyse syntaxique pour dresser le graphe des dépendances et ainsi pouvoir orienter directement la résolution de toute équation en débutant par les "feuilles" : par les équations les moins dépendantes mais relevantes pour le problème.

BIBLIOGRAPHIE

A practical introduction to denotational semantics
Lloyd Allison
Cambridge University Press, 1986.

A universal Top-Down Fixpoint Algorithm
Baudouin Le Charlier, Pascal Van Hentenryck
University of Namur

Compilateurs. Principes, techniques, outils
Alfred Aho, Ravi Sethi, Jeffrey Ullman
InterEditons, Paris, 1989.

La fiabilité des programmes
H. Leroy
photocopies

L'analyse statique des programmes par interprétation abstraite
Baudouin Le Charlier
Université de Namur

Le langage C++
Bjarne Stroustrup
InterEditions, Paris, 1989

Mathematical Theory of Computation
Zohar Manna
McGRAW-HILL, New York, 1974.

Mathématiques discrètes
Michel Marchand
De Boeck, Bruxelles, 1989.

Théorie des programmes
LIVERCY
Bordas, Paris, 1978.

ANNEXE

FONDEMENTS THEORIQUES

THEORIE DU POINT FIXE

La notion d'ensemble

Un ensemble est une collection d'objets tel qu'il existe une définition (appelée fonction caractéristique) de cet ensemble, c'est-à-dire qu'il y a toujours moyen de décider si un objet quelconque fait partie ou non de l'ensemble.
Ce qui caractérise un ensemble est sa loi d'appartenance : $\in$.

Un ensemble peut être défini en intention ou en extension. Une définition en intention (ou en compréhension) énonce des caractéristiques, des propriétés des éléments tel que chaque élément possède chacune des propriétés et tel que la possession de chacune des propriétés assure l'appartenance de cet élément à l'ensemble. Une définition en extension est une énumération exhaustive des éléments de l'ensemble.

Ainsi l'ensemble de entiers naturels inférieurs à 4 peut-il être défini en intention par $\{x|x\in\mathbb{N} \wedge x<4\}$, en extension par $\{1,2,3\}$. Ces deux définitions recouvrent les mêmes objets ; pourtant la première comporte - à première vue - plus de renseignements ; ceux-ci sont "cachés" dans la seconde.

Si l'on munit un ensemble des opérateurs binaires $\cup,\cap$, de l'opérateur unaire de complément -, de l'ensemble vide $\emptyset$ et de l'ensemble universel désigné ici par 1, on forme une algèbre booléenne, dont voici les propriétés essentielles :

idempotence	$x\cap x=x$	$x\cup x=x$
commutativité	$x\cap y=y\cap x$	$x\cup y=y\cup x$
associativité	$x\cap(y\cap z)=(x\cap y)\cap z$	$x\cup(y\cup z)=(x\cup y)\cup z$
distributivité	$x\cap(y\cup z)=(x\cap y)\cup(x\cap z)$	$x\cup(y\cap z)=(x\cup y)\cap(x\cup z)$
neutre	$x\cap 1=x$	$x\cup\emptyset=x$
absorbant	$x\cap\emptyset=\emptyset$	$x\cup 1=1$
complément	$x\cap -x=\emptyset$	$x\cup -x=1$

Ces propriétés nous serviront de définitions de $\cup$ et $\cap$; mais on peut encore les définir par :

$$x\cup y = \{z \mid z\in x \vee z\in y\}$$
$$x\cap y = \{z \mid z\in x \wedge z\in y\}$$

La notion de produit cartésien

Le produit cartésien de deux ensembles est l'ensemble de tous les couples que l'on peut former à partir d'un élément du premier ensemble et un élément du second ensemble. On désignera par A*B le produit cartésien de A par B.

$$A*B = \{ \langle x,y\rangle \mid x\in A \wedge y\in B \}$$

On remarquera que le produit cartésien de produits cartésiens tel que (A*B)*(C*D) est en relation bijective (un à un) avec A*(B*C)*D ou ((A*B)*C)*D ; il est donc concevable de ne pas mentionner les parenthèses : $(A*B)*(C*D) \equiv A*B*C*D$

La notion de relation

Une relation R d'un ensemble A vers un ensemble B est une partie du produit cartésien A*B et donc un ensemble de couples : $R \subseteq A*B$.
On désigne souvent le couple <a,b> dans la relation R par aRb.
Une relation R étant un ensemble (de couples), on peut définir R comme on définit un ensemble : par compréhension ou énumération.

La notion de fonction

Une fonction f est une relation (donc une partie d'un produit cartésien) telle que $f \subset A*B \land (\forall a \in A \land \forall b,c \in B ((afb) \land (afc)) \Rightarrow (b=c))$
ou encore : $(\langle a,b \rangle, \langle a,c \rangle \in f) \Rightarrow (c=b)$

On désigne généralement une fonction par une minuscule, une relation par une majuscule et au lieu d'écrire afb, on écrit f(a)=b

On désigne une fonction par sa signature :
$f : A \dashrightarrow B : x \mapsto f(x)$

Ci-dessus, A est le domaine de la fonction, B son codomaine.

Une fonction f étant un ensemble (de couples), on peut définir f comme on définit un ensemble : par compréhension ou énumération. On excluera toutefois pour l'instant toute définition récursive (directe ou indirecte) car nous n'en avons pas encore donné la signification.

Syntaxe et sémantique

Toute fonction est généralement représentée comme :
f : A --> R
où f est l'identificateur de la fonction
A est le type de l'argument
R est le type du résultat.

Une fonction est "appelée" pour qu'un argument soit mis en rapport avec un résultat.

On doit distinguer dans la définition d'une fonction :
- la syntaxe de la fonction :
la syntaxe énonce les règles d'usage de la fonction (les règles d'écriture de l'appel) ; on ne s'y intéressera pas ici.
- sa sémantique :
la sémantique énonce la signification d'une formulation syntaxique correcte : quelles sont les valeurs d'argument et de résultat que la fonction formulée met en rapport. C'est ce qui nous intéresse.

Une fonction énonce une relation entre des éléments de A et des éléments de D telle qu'à un élément de A corresponde au plus un élément de D.

Pour définir cette relation on peut :

1) présenter une table en deux colonnes, au niveau de chaque ligne de la table on présentera dans la première colonne une valeur de A et dans la seconde une valeur de R telle qu'il n'existe qu'au plus une ligne comportant une certaine valeur pour A. Une telle table n'est réalisable que pour les fonctions qui ne sont définies que pour un ensemble limité de valeurs d'argument. C'est la présentation "bases de

données", elle a l'avantage de présenter une fonction comme une partie du produit cartésien : $f \subseteq A*R$

2) on peux également proposer une <u>méthode de calcul</u> qui soit équivalente à la table ci-dessus : un algorithme qui calcule pour chaque valeur de la première colonne d'une ligne quelconque de la table la valeur qui est en seconde colonne. Cette présentation a l'avantage d'être brève. Par contre, il n'est pas toujours évident de démontrer son équivalence avec la table ; de plus, elle peut cacher le fait que d'autres algorithmes pourraient être plus performants : il y a surdéfinition. Cette sémantique est appelée <u>opérationnelle</u> ou <u>procédurale</u>.

3) on peux encore définir la fonction par des formules de la logique des prédicats. On énonce d'une part la <u>précondition</u> c'est-à-dire la formule logique (proposition) que doit respecter l'argument pour que la fonction soit définie ; d'autre part la <u>postcondition</u> c'est-à-dire la formule logique qui fait correspondre une valeur de résultat à la valeur d'argument. Cette présentation a l'avantage d'éliminer les détails de l'implémentation. Cette sémantique est appellée <u>axiomatique</u>.

4) enfin une fonction peut être définie dans le cadre d'une théorie générale des fonctions où chaque fonction est un objet. Dans l'ensemble des fonctions, chaque objet-fonction aura certaines caractéristiques et entrera en relation avec d'autres fonctions. Nous allons exposer et exploiter cette "vision" qu'on appelle la sémantique dénotationnelle car elle donne un sens très précis à la récursivité.

Fonctions totales et fonctions partielles

Une fonction est dite <u>totale</u> ssi $\forall a \in A\ \exists b \in B \mid f(a)=b$
ou encore f est totale $\Leftrightarrow\ \forall a \in A\ \exists \langle a,b\rangle \in f$
sinon elle est dite <u>partielle</u>,

On admettra les écritures suivantes :

$A \rightharpoonup B$	une fonction partielle de A dans B
$A \rightarrow B$	une fonction totale de A dans B
$[\![A \rightarrow B]\!]$	l'ensemble des fonctions totales de A dans B
$A*B$	l'ensemble des couples du produit cartésien ce qui donne : $A \rightarrow B \subset A*B$
$f \in [\![A \rightarrow B]\!]$	f est une fonction totale de A dans B
$\mid x \mapsto f(x)$	telle que à toute valeur x du domaine de la fonction, la fonction associe la valeur $f(x) \in B$ autrement dit : $\forall x \in A\ \langle x,f(x)\rangle \in f$

Soit f une fonction partielle de A dans B, soit a un élément de A pour laquelle la fonction f n'est pas définie ; informatiquement parlant, l'appel de f(a) ne peut donner de solution : f doit "boucler" indéfiniment.

Exemples : - la fonction "suivant" de $\mathbb{N}$ dans $\mathbb{N}$ est définie pour toute valeur entière de l'argument : elle est totale.

suivant $\in [\![\mathbb{N} \rightarrow \mathbb{N}]\!] \mid x \mapsto x+1$

- la fonction "précédent" de $\mathbb{N}$ dans $\mathbb{N}$ n'est pas définie pour la valeur 0 de l'argument : elle est partielle ; mais précédent sera totale par

précédent $\in [\![\mathbb{N} \backslash \{0\} \rightarrow \mathbb{N}]\!] \mid x \mapsto x-1$

Notion de transformation

Appelons transformation une fonction dont les domaines et codomaines sont identiques : $f : A \to A$.

Notion mathématique de point fixe

Un point fixe d'une transformation f est une valeur x du domaine de la fonction telle que $f(x)=x$. Une transformation peut avoir plusieus points fixes ou aucun. La fonction d'identité a une infinité de points fixes ; la fonction 'incrément' n'en a pas et la fonction $f(x)=-x$ n'en a qu'un.

Relation d'ordre sur les fonctions de même signature

Les fonctions qui ont même signature constituent un ensemble ; cet ensemble peut être muni d'une relation d'ordre comme suit :

$$f \sqsubseteq g \Leftrightarrow (f,g \in [\![A \to B]\!]) \wedge$$
$$(X=\{a \in A \mid \exists b \in B \mid f(a)=b\}) \wedge (X \subseteq \{a \in A \mid \exists b \in B \mid g(a)=b\}) \wedge$$
$$(\forall a \in X\ f(a)=g(a))$$

On vérifiera aisément qu'il s'agit d'une relation d'ordre partielle (réflexivité, antisymétrie, transitivité).
Si $f \sqsubseteq g$ on dira que g est au moins aussi définie que g ou que f est moins définie que g.

Notion de convergence

Au regard de la relation d'ordre définie ci-dessus, on peut concevoir une suite (souvent infinie) de fonctions f_i telle que $f_1 \sqsubseteq f_2 \sqsubseteq f_3 \sqsubseteq f_4 \sqsubseteq \ldots$

On dira que cette suite converge vers f si pour tout x tel que f(x) est défini, il existe n tel que $f_n(x)=f(x)$.

Cette suite de fonction est intéressante car elle "atteint" f(x) par approximations successives.

Notion d'extension stricte

On peut faire en sorte que toutes nos fonctions soient TOTALES, ceci grâce à l'élément $\perp$ que l'on ajoutera à son codomaine :

Soit $f \in [\![A \to B]\!]$
Définissons $f^+ \in [\![A \to B \cup \{\perp\}]\!]$ tel que
si $X=\{a \in A \mid \exists b \in B \mid f(a)=b\}$
alors $(\forall a \in X\ f^+(a)=f(a)) \wedge (\forall a \in A \setminus X\ f^+(a)=\perp)$

On appelera f^+ définie comme ci-dessus l'extension stricte de f.

On peut donc redéfinir $\sqsubseteq$ comme :
si $f,g \in [\![A \to B]\!]$
$f \sqsubseteq g \Leftrightarrow \forall x\ f(x)=\perp \vee f(x)=g(x)$

Notion de domaine primitif

Si nous décidons de compléter tous nos ensembles d'un élément $\perp$, nous munissons nos domaines d'une relation d'ordre définie par

$\forall\ x,y \in A^+ \quad x \preceq y \Leftrightarrow \quad x=\perp \vee x=y$

Appelons domaine primitif tout domaine qui vérifie la proposition ci-dessus.

Soit $f \in [\![A \rightarrow R]\!]$
Soit $R^+ = R \cup \{\perp\}$
On définit f^+ par
$f^+ \in [\![A \rightarrow R^+]\!] \quad | \ x \mapsto f^+(x)$ défini par
$<x,y> \in f \Rightarrow \quad f(x)=f^+(x)$
$<x,y> \notin f \Rightarrow \quad f^+(x)=\perp$

Exemple : soit R={vrai,faux}
R^+={$\perp$,vrai,faux}
$\preceq$={<$\perp$,$\perp$>,<$\perp$,vrai>,<$\perp$,faux>,<vrai,vrai>,<faux,faux>}
soit $\mathbb{N}$ l'ensemble des entiers positifs
$\mathbb{N}^+=\mathbb{N}\cup\{\perp\}$
$\preceq=\{<\perp,\perp>\}\cup\{<\perp,x>|x\in\mathbb{N}\}\cup\{<x,x>|x\in\mathbb{N}\}$

Par la suite, on entendra par "domaine primitif" tout ensemble créé à partir d'un domaine prédéfini tel que "INTEGER", "REAL", "STRING" etc auquel on aura adjoint - pour les seuls besoins de la sémantique - un élément fictif distinct de tout autre et noté $\perp$.

Un domaine primitif peut, théoriquement, compter un nombre infini d'éléments.

Le symbole $\perp$ représente l'absence de valeur ; il n'est là que pour que toute fonction soit totale. L'élément $\perp$ d'un domaine primitif n'a pas besoin d'être "implémenté", il ne va servir qu'à exprimer la sémantique d'un programme. Toutefois on devra s'inquiéter de la signification d'une fonction qui reçoit comme argument $\perp$.

On vérifiera aisément que $\preceq$ est réflexive, antisymétrique et transitive. On remarquera que $\preceq$ est partielle : vrai et faux ne sont pas en relation par $\preceq$; deux entiers ne sont pas en relation par $\preceq$: $\forall x,y\in\mathbb{N}\ x\neq y \Rightarrow \quad (x\npreceq y \wedge y\npreceq x)$

R^+ ou tout domaine primitif ainsi construit à partir de n'importe quel ensemble, fini ou infini, est un ensemble partiellement ordonné : partially ordered set ou pos.

Quelque soit R^+ domaine primitif, $\perp$ est "son plus petit" élément :
$\forall x\in R^+ \ \perp\preceq x$

Remarque : chaque domaine primitif aura "son" plus petit élément et donc "sa" relation d'ordre. Pour alléger la lecture, on acceptera cependant de désigner chacun de ces plus petits éléments par le même symbole $\perp$, et chaque relation d'ordre par le même symbole $\preceq$. Lorsqu'il faudra bien distinguer les $\perp$, on les indicera de l'ensemble d'appartenance.

$\preceq_N=\{<\perp_N,\perp_N>\}\cup\{<\perp_N,x>|x\in\mathbb{N}\}\cup\{<x,x>|x\in\mathbb{N}\}$

Relation d'ordre partiel sur les fonctions

Comme l'argument d'une fonction peut être un appel de fonction, comme R^+ peut à son tour devenir le domaine d'une fonction, il nous faut savoir ce que peut donner $f(\perp)$.

Intuitivement, une fonction qui reçoit en paramètre quelque chose qui n'est pas défini doit "se planter" s'il y a passage de paramètre par valeur alors que s'il y a appel de paramètre par nom, elle ne se plantera que s'il y a effectivement évaluation du paramètre.

Soit la fonction constante :

$f \in [\![\mathbb{N} \to \mathbb{N}]\!] : x \mapsto 3$

On peut lui envisager deux 'extensions' :

$f_1^+ \in [\![\mathbb{N}^+ \to \mathbb{N}^+]\!] \mid x \mapsto 3$

$f_2^+ \in [\![\mathbb{N}^+ \to \mathbb{N}^+]\!] \mid x \mapsto$ <u>si</u> $x=\perp$ <u>alors</u> $\perp$ <u>sinon</u> 3

La fonction $f1^+$ peut être qualifiée de "davantage définie" que $f2^+$.

On dira que f <u>est moins définie</u> que g ou $f \preceq g$ comme suit :

$f \in [\![A^+ \to R^+]\!] \mid x \mapsto f(x)$

$g \in [\![A^+ \to R^+]\!] \mid x \mapsto g(x)$

$f \preceq g \Leftrightarrow \forall x \in A^+ \quad (f(x)=g(x)) \vee (f(x)=\perp)$

$\preceq$ est une relation d'ordre partiel dans l'ensemble des fonctions de A^+ dans R^+. On vérifiera aisément que $\preceq$ est

- réflexive $\quad \forall f \in [\![A^+ \to R^+]\!] \; f \preceq f$
- antisymétrique $\quad \forall f,g \in [\![A^+ \to R^+]\!] \; f \preceq g \wedge g \preceq f \Rightarrow f=g$
- transitive. $\quad \forall f,g,h \in [\![A^+ \to R^+]\!] \; f \preceq g \wedge g \preceq h \Rightarrow f \preceq h$

On peut encore définir de la manière suivante la relation "moins défini que " :

$f,g \in [\![A^+ \to R^+]\!] \Rightarrow f \preceq g \Leftrightarrow \forall x \in A^+ \; f(x) \sqsubseteq g(x)$

Il est à noter que seules certaines fonctions sont comparables entre elles : la relation d'ordre est partielle.

Soient deux fonctions telles que $f \preceq g$, on dira que f est une <u>restriction</u> de g, ou que g est une <u>extension</u> de f.

Deux fonctions f et g seront sémantiquement équivalentes si

$f,g \in [\![A^+ \to R^+]\!] \wedge f \preceq g \wedge g \preceq f$

Notion d'algorithme

Un algorithme est une <u>solution à une classe de problèmes, exécutable par un "processeur" et exprimée dans son langage</u>.

Les définitions ci-dessus sont "mathématiques", c'est-à-dire théoriques, exprimées indépendamment de tout processeur (les mathématiciens ont définis a+b pour tout a et b entier, pourtant personne n'a dû faire les calculs). En informatique, on manipule des algorithmes ; mais un algorithme est un schéma qui sera traduit dans le langage bien concret d'un processeur bien concret qui devra effectuer concrètement le calcul d'une fonction pour une certaine valeur bien concrète de ses arguments.

On doit distinguer deux aspects d'un algorithme :

- sa définition qui est "abstraite"
- son exécution qui est "concrète".

La définition d'un algorithme est la définition "en compréhension" d'une fonction. L'ensemble des exécutions possibles d'un algorithme est la définition "en extension" de cette même fonction.

Un algorithme est une <u>fonction</u> et non une relation car deux exécutions d'un algorithme pour les mêmes valeurs d'arguments doivent retourner les mêmes résultats.

Un algorithme est une fonction <u>partielle</u> car lorsqu'il est appelé hors de son domaine de validité, il boucle : il refuse de donner un résultat qui serait erroné (écartons les problèmes de panne).

Si un algorithme est défini par la fonction mathématique (totale) qui le représente on peut établir une relation entre l'ensemble des algorithmes et l'ensemble des fonctions. Dans cette relation, on ne s'intéresse qu'aux résultats des exécutions de l'algorithme : le "comment" n'a pas d'intérêt (sinon plus tard pour 'optimiser'). Deux algorithmes distincts peuvent donc être représentés par la même fonction : la même table au sens où une fonction est un ensemble de couples.

On peut définir une relation d'ordre sur les algorithmes au regard de la relation d'ordre sur les fonctions ; certains algorithmes seront donc "davantage définis que d'autres" ; on pourrait croire qu'un algorithme "moins défini" est "plus rapide", "moins exigeant" puisqu'il a "moins à faire" mais ce n'est évidemment pas une loi et de toute façon, ces considérations sont absentes dans l'analyse des fonctions.

Notion de monotonie des fonctions informatiques

Qu'un algorithme doit être "exécuté" implique que son exécution doit s'effectuer avec des ressources (temps et espace-mémoire) finies ; cette finitude des algorithmes va imposer aux fonctions qui les représentent le respect de certaines propriétés que l'on va maintenant définir : la monotonie et la continuité.

Soit la fonction f définie par

$f \in [\![N^+ \rightarrow N^+]\!] \mid x \mapsto$ si $x=\perp$ alors 1 sinon $\perp$

Cette fonction est pour le moins surprenante car - informatiquement parlant et au sens où nous avons introduit $\perp$ - elle doit "se planter" quand l'évaluation de son argument "ne se plante pas" et "ne pas se planter" quand l'évaluation de son argument "se plante".

Pour exprimer que nos fonctions ne peuvent retourner un résultat davantage défini quand elles recoivent des données moins définies, on va dire que nos fonctions doivent être monotones :

$f \in [\![A \rightarrow B]\!]$ est <u>monotone</u> ssi $\forall x,y \in A \; x \preceq y \Rightarrow f(x) \preceq f(y)$

On dit encore que f est <u>croissante</u> mais il vaudrait mieux dire <u>non décroissante</u>.

Toute fonction constante est une fonction monotone.
La fonction suivante n'est pas monotone :

$f \in [\![R \rightarrow \{$<u>vrai</u>,<u>faux</u>$\}]\!] \mid x \mapsto$ <u>si</u> $x=\perp$ <u>alors</u> <u>vrai</u> <u>sinon</u> <u>faux</u>

en effet $\perp \preceq x$ alors que <u>vrai</u>$\not\preceq$<u>faux</u>

<u>Nous ne traiterons plus que de fonctions totales et monotones définies sur des domaines primitifs.</u>

Nous avons décidé de n'accepter comme fonctions que les fonctions monotones, or les schémas algorithmiques sont des articulations de fonctions ; vérifions que les déclarations de fonctions par combinaison de fonctions *monotones* donnent bien des fonctions monotones.

Dans la définition d'une fonction
$f \in [\![A \to B]\!] \mid x \mapsto f(x)$
la valeur f(x) est définie par une combinaison de "primitives" et/ou de fonctions ailleurs définies, par exemple :
$f \in [\![A \to B]\!] \mid x \mapsto g(x)$
$f \in [\![A \to B]\!] \mid x \mapsto h(g(x))$
$f \in [\![A \to B]\!] \mid x \mapsto$ si $p(x)$ alors $g(x)$ sinon $h(x)$.

On traitera plus tard de la récursivité par exemple dans
$f \in [\![A \to B]\!] \mid x \mapsto$ si $p(x)$ alors $g(x)$ sinon $f(h(x))$
ou plus subtilement dans
$f \in [\![A \to A]\!] \mid x \mapsto$ si $p(x)$ alors $g(x)$ sinon $k(x)$
$k \in [\![A \to A]\!] \mid x \mapsto$ si $q(x)$ alors $r(x)$ sinon $f(x)$
ainsi que des fonctions transmises en paramètre, par exemple dans
$f \in [\![A*B \to C]\!] \mid (a,b) \mapsto b(g(a))$
le domaine des fonctions est pour l'instant exclu.

Tout ce qu'on peut trouver à droite de $\mapsto$ ce sont
- des "variables" (paramètre formel) énoncées à gauche de $\mapsto$ et dont la signification est qu'elles seront remplacées par les valeurs des paramètres d'appels de la fonction (paramètre effectif)
- des combinaisons de fonctions pour l'instant non récursives.

On règlera immédiatement le cas des "primitives" c'est-à-dire des fonctions prédéfinies en disant tout simplement que ces fonctions sont garanties correctes ; seule nous intéresse l'expression de la sémantique des fonctions que nous créons nous-même par combinaison de primitives.
Donc la sémantique de la fonction
$f \in [\![A \to B]\!] \mid x \mapsto g(x)$
avec g une primitive définie par
$g \in [\![C \to D]\!] \mid x \mapsto g(x)$
est évidente pour autant que l'on ait $A \subseteq C$ et $D \subseteq B$:
$f \in [\![A \to B]\!] \mid x \mapsto$ si $x \in C$ alors $g(c)$ sinon $\perp$
On vérifie aisément que cette définition est monotone.

La fonction *si alors sinon* est un cas classique de fonction à plusieurs arguments. Est-elle monotone ? Tout dépend de sa définition sémantique. La suivante est monotone :

si $\perp$ *alors* b *sinon* c $\Rightarrow$ $\perp$ quelque soit b et c
si *vrai* *alors* b *sinon* c $\Rightarrow$ b quelque soit c
si *faux* *alors* b *sinon* c $\Rightarrow$ c quelque soit b

Ce qu'on peut encore écrire :
if(a,b,c) $\triangleq$ *si* $a=\perp$ *alors* $\perp$ *sinon* *si* a=vrai *alors* b *sinon* c

Les termes si alors sinon soulignés sont des opérateurs du métalangage d'expression de la sémantique du if.

Donnons un sens à f(g(x)) :
si $f \in [\![A \to B]\!] \mid x \mapsto f(x)$
et $g \in [\![B \to C]\!] \mid x \mapsto g(x)$
alors $f(g(x)) \between$ si $x=\perp$ alors $\perp$ sinon $f(g(x))$

Cette écriture peut paraître choquante dans la mesure où le f(g(x)) de droite n'est pas défini ; en fait il ne s'agit pas du f(g(x)) en train

d'être défini mais bien d'une fonction définie dans le métalangage : "appeler la fonction f avec comme argument la valeur g(x) sachant que x n'est pas $\perp$; comme x n'est pas $\perp$, l'écriture g(x) n'est pas ambigue".

On peut aisément vérifier que notre imbrication de fonction f(g(x)) est monotone si f et g sont monotones ; bien entendu l'écriture f(f(x)) ou $f^n(x)$ est autorisée.

On peut considérer qu'une mise en séquence de deux fonctions $f^\circ g$ signifie "exécuter g sur les données retournées par f" ce qui s'écrira : $^\circ(f,g,x) = g(f(x))$
la fonction monotone $^\circ$ s'écrira :

$$^\circ(f,g,x) \triangleq \text{si } f(x)=\perp \text{ alors } \perp \text{ sinon } g(f(x)).$$

Relation d'ordre $\preceq$ partiel dans les produits cartésiens

Les domaines de nos fonctions ont été jusqu'à présent des "domaines primitifs", on peut maintenant accepter des domaines plus complexes ; commençons par les produits cartésiens.

Soit le domaine A*B, on dira que

$$\forall\langle x,y\rangle,\langle z,w\rangle \in A*B \quad \langle x,y\rangle \preceq \langle z,w\rangle \Leftrightarrow x \preceq z \wedge y \preceq w.$$

Si l'on considère que, par définition, A*B*C = (A*B)*C, on étend l'ordre partiel $\preceq$ à tous les produits cartésiens d'ensembles ordonnés par $\preceq$.

Nos domaines primitifs ont chacun un plus petit élément noté $\perp$; en effet : $\forall x \in A \ \perp_A \preceq x$
Il en est de même des produits cartésiens de domaines primitifs :

$$\forall\langle x,y\rangle \in A*B \quad \perp_{A*B} = \langle \perp_A, \perp_B\rangle \preceq \langle x,y\rangle$$

On démontre aisément, par récurrence, que tous les domaines appartenant à $D \triangleq \{ P \mid P \text{ est un domaine primitif} \} \cup \{ A*B \mid A,B \in D \}$ sont partiellement ordonnés et ont un plus petit élément. Désormais, on appelera D l'ensemble des produits cartésiens finis de domaines primitifs.

Admettons que toute fonction "à plusieurs arguments" est une fonction à un argument qui est un produit cartésien.

On définira maintenant l'extension stricte comme

$$f \in [\![A*B \rightarrow C]\!] \mid (a,b) \mapsto f(a,b)$$
$$f^+ \in [\![A^+ \rightarrow B^+]\!] \mid \text{si } a=\perp \text{ ou } b=\perp \text{ alors } \perp \text{ sinon } f(a,b)$$

On vérifiera aisément qu'au regard de toute fonction partielle à plusieurs arguments, l'extension stricte est toujours la moins définie des extensions.

Comme on définit une relation d'ordre $\preceq$ pour les produits cartésiens, on peut directement étendre notre définition de la monotonie des fonctions dans les produits cartésiens finis de domaines récursifs.

$$f \in [\![A \rightarrow B]\!] \text{ est } \underline{\text{monotone}} \text{ ssi } \forall x,y \in A \ x \preceq y \Rightarrow f(x) \preceq f(y)$$

avec A et B produits cartésiens finis de domaines primitifs (on exclut pour l'instant que A soit un domaine fonctionnel)

Une fonction en paramètre

Une fonction transmise en paramètre est un objet qui peut être infini : défini pour un ensemble infini de valeurs.

Intuitivement, on peut dire que si une fonction est transmise en paramètre, celle-ci ne peut être consultée qu'un nombre fini de fois. Ce sera le cas dans

$f \in [\![A*[\![A \to A]\!]]\!] \to A]\!] \mid (x,g) \mapsto g(g(x))$

et encore faut-il être sûr que g ne fait pas appel directement ou indirectement à f.

Soit une fonction f définie sur la fonction h ; ces deux objets peuvent être infinis MAIS quand il sera effectué f(x), le calcul de f(x) ne peut dépendre que d'une partie finie de la table des correspondances de h.

Pour donner une signification aux définitions de fonction où une fonction est donnée en paramètre, la monotonie des fonctions n'est plus suffisante, il nous faut maintenant introduire les notions d'espace inductif et de continuité.

Notions d'éléments comparables

Nous avons équipé nos domaines d'une relation d'ordre $\preceq$.

Rappelons qu'une relation d'ordre $(A,\leq)$ est une fonction partielle :

$\leq \in [\![A*A \rightharpoonup \{VRAI,FAUX\}]\!]$

Deux éléments quelconques d'un ensemble ordonné ne sont pas nécessairement "comparables" ainsi dans $(P(E),\subseteq)$ avec P(E) les parties de l'ensemble E, si $E=\{1,2,3,4\}$, $\{1,2\}$ et $\{2,3\}$ ne sont pas comparables puisqu'aucun n'inclut l'autre.

On dit que a et b sont comparables ssi $(a\leq b) \vee (b\leq a)$.
Quand des éléments ne sont pas comparables, on dit que l'ensemble est partiellement ordonné.
Quand tous les éléments sont comparables, on dit que l'ensemble est totalement ordonné (on dit aussi linéaire)

Dans un ensemble totalement ordonné $(A,\leq)$, on a

$a<b \Leftrightarrow \neg(b\leq a)$

c'est le cas de $(\mathbb{N},\leq)$, ce n'est pas le cas de $(\mathbb{N},\preceq)$ où $2\preceq 3$ n'est pas défini.

Notion de chaîne

Appelons chaîne (x_i) toute suite croissante finie ou infinie d'éléments d'un ensemble partiellement ordonné :

$(X,\preceq)$ ensemble partiellement ordonné
$\forall i\geq 0 \; x_i \in X$
$x_0 \preceq x_1 \preceq x_2 \preceq x_3 \preceq x_4 \preceq \ldots$

Il est à noter qu'en ce qui concerne les domaines primitifs, toute chaîne infinie d'éléments du domaine de la forme

$x_0 \preceq x_1 \preceq x_2 \preceq x_3 \preceq x_4 \preceq \ldots$

est stationnaire c'est-à-dire que $\exists i\geq 0 \mid \forall j>i \; x_i=x_j$.
En effet, cette chaîne ne peut se composer que d'au maximum deux valeurs distinctes.

Si l'on définit $\prec$ par $x \prec y \Leftrightarrow x \preceq y \wedge x \neq y$
on peut encore écrire que toute chaîne de la forme $x_0 \prec x_1 \prec x_2 \prec x_3 \prec \ldots$ est finie ou comporte un nombre fini d'éléments.

Si le domaine des éléments de la chaîne $x_0 \preceq x_1 \preceq x_2 \preceq x_3 \preceq x_4 \preceq \ldots$ est un <u>produit cartésien fini de domaines primitifs</u> A*B*C*..., cette chaîne sera également stationnaire tout simplement par ce que le domaine A*B*C*... articule un nombre fini de domaines primitifs.
On démontre aisément par induction sur le nombre de domaines primitifs que si le domaine est un produit cartésien de n domaines primitifs la chaîne $x_0 \preceq x_1 \prec x_2 \prec x_3 \prec \ldots$ ne peut se composer que d'au maximum n+1 éléments.

Dans le domaine des fonctions, du fait qu'un domaine primitif peut se composer d'une infinité d'éléments (comme dans $\mathbb{N}^+$), on peut construire des chaînes composées d'une suite infinie d'éléments distincts, il en serait de même dans le domaine des produits cartésiens infinis de domaines primitifs.

Mais le domaine des produits cartésiens infinis de domaines primitifs ne peut être un objet informatique et on n'appelle jamais une fonction que pour une valeur particulière de son domaine : il ne faut donc pas ordonner les fonctions qui calculent l'addition d'entiers quelconques mais les fonctions qui calculent l'addition de deux valeurs d'entiers particulières. Le quantificateur $\forall$ disparaît :

$f,g \in [\![\mathbb{N}^+ * \mathbb{N}^+ \rightarrow \mathbb{N}^+]\!] : (a,b) \mapsto a+b$
$a,b \in \mathbb{N}^+$
$f \sqsubseteq g \Leftrightarrow f(a,b)=\perp \vee f(a,b)=g(a,b)$

La plus longue chaîne sera de deux éléments distincts !

<u>Suite convergente de fonctions</u>

On se souvient, qu'une fonction peut être approximée par une suite qui converge vers f :

$\perp \sqsubseteq f_1 \sqsubseteq f_2 \sqsubseteq f_3 \sqsubseteq f_4 \sqsubseteq \ldots$ avec f_i telle que $f_i \sqsubseteq f$:

cette suite converge vers f si pour tout x tel que f(x) est défini, il existe n (fini) tel que $f_n(x)=f(x)$.

On dira que la suite de fonctions $\langle f_i \rangle$ <u>converge</u> vers la fonction f par

$\langle f_i \rangle \subseteq [\![A \rightarrow B]\!]$
$f \in [\![A \rightarrow B]\!]$
$\langle f_i \rangle$ converge vers f ssi
$\forall i \geq 0 \; f_i \preceq f_{i+1} \wedge \forall a \in A \; \exists i \geq 0 \mid f_i(a)=f(a)$

Cette notion de convergence exprime bien que si l'on parvient à définir abstraitement la suite $\langle f_i \rangle$ comme convergente à f, pour toute valeur de x, la valeur de f(x) peut être obtenue par un nombre fini d'approximations successives (n est fini).

Reprenons notre problème : celui d'une fonction qui est transmise en paramètre à une fonction. On a dit que cette fonction paramètre ne peut être consultée qu'un nombre fini de fois ; mais à chaque fois qu'elle est consultée, on en sait un peu plus ; autrement dit, on constitue une chaîne de fonctions qui approxime la fonction recherchée. On arrêtera la consultation quand on en connaîtra suffisamment de la fonction : quand en savoir plus n'apporte rien de nécessaire au problème.

Notion d'espace inductif

La notion d'espace inductif va permettre de considérer des ensembles où toute chaîne n'est pas nécessairement stationnaire, peut se composer d'une infinité d'éléments distincts mais au moins définit un unique élément qui appartient à l'ensemble.

On appellera espace inductif tout ensemble E (non vide) partiellement ordonné par $\preceq$ qui vérifie :

$\exists \bot \in E \mid \forall x \in E \;\bot \preceq x$
$\forall x_0, x_1, x_2, x_3, x_4, \ldots \in E \mid x_0 \preceq x_1 \preceq x_2 \preceq x_3 \preceq x_4 \preceq \ldots$
$\exists e \in E \mid \forall i \geq 0 \; x_i \preceq e \wedge (\forall g \in E \; (\forall i \geq 0 \; x_i \preceq g) \Rightarrow e \preceq g)$

en français : toute chaîne (suite croissante) est bornée par un élément de l'ensemble.

On dira que tout ensemble E est inductif pour la relation d'ordre partiel $\preceq$ définie sur cet ensemble si $(E, \preceq)$ est un espace inductif.
En anglais : complete partial order ou cpo.

On appellera lub ou borne supérieure d'une chaîne, cet élément de l'ensemble qui est le plus petit des éléments de l'ensemble qui sont plus grands que tout élément de la chaîne.
En mathématique, on parle plus souvent de supremum : le plus petit (ou minimum) des majorants.

On représentera désormais $x_0 \preceq x_1 \preceq x_2 \preceq x_3 \preceq x_4 \preceq \ldots$ par $\langle x_i \rangle$:

$\langle x_i \rangle \triangleq \{ x_i \mid i \geq 0 \wedge x_0 \preceq x_1 \preceq x_2 \preceq x_3 \preceq x_4 \preceq \ldots \}$

et on désignera le lub par $lub\langle x_i \rangle$

Remarques :
- le lub n'appartient pas nécessairement à la chaîne.
- le lub d'une chaîne stationnaire est son plus grand élément et donc appartient à la chaîne.

Exemples :

Nos "produits cartésiens finis de domaines primitifs" sont des espaces inductifs ; le lub de toute chaîne de nos "produits cartésiens finis de domaines primitifs" appartient à la chaîne puisque toute chaîne y est stationnaire.

$(\mathbb{N}, \leq)$ n'est pas un espace inductif bien que $\leq$ soit une relation d'ordre puisqu'on peut toujours prolonger toute chaîne.

Soit un ensemble E fini ou non, soit $\mathscr{E}$ l'ensemble des parties de E ; $\mathscr{E}$ est un espace inductif avec $\preceq \equiv \subseteq$ et $\bot \equiv \emptyset$ car toute chaîne bien que pouvant être infinie sera toujours bornée par $E \in \mathscr{E}$.

L'ensemble des intervalles dans $\mathbb{N}$ ou $\mathbb{R}$ est inductif pour la relation d'inclusion $\subseteq$.

L'ensemble des nombres réels de l'intervalle [0,1] est un espace inductif pour $\leq$ mais pas l'ensemble des nombres réels de l'intervalle [0,1[.

Dans l'ensemble des nombres rationnels, la chaîne infinie des nombres qui définissent PI a bien une borne supérieure (PI) mais PI n'est pas un nombre rationnel.

Espaces strictement inductifs

L'intervalle [0,1] dans $(\mathbb{R},<)$ est inductif mais on peut construire une chaîne infinie d'éléments distincts telle que son lub existe dans [0,1] mais n'appartienne pas à la chaîne.

Pour distinguer les espaces inductifs tels que toute chaîne $\langle x_i \rangle$ comporte son lub ou non (et la suite $x_0 \prec x_1 \prec x_2 \prec x_3 \prec \ldots$ sera finie ou non), on parlera d'espace strictement inductif.

Nos produits cartésiens finis de domaines primitifs sont strictement inductifs (confer plus haut - chaîne stationnaire).

Si nous parvenons à construire un domaine de fonctions qui est espace inductif alors

- si cet espace est strictement inductif, toute chaîne sera stationnaire et donc toute fonction ne peut être "dernier élément" que de chaînes qui se composent d'un nombre finis d'éléments : on atteindra cette fonction par un nombre fini d'approximations.
- si cet espace est simplement inductif toute fonction pourra être atteinte comme la limite d'une chaîne non stationnaire : on atteindra cette fonction par un nombre infini d'approximations.

Intuitivement, une fonction telle que

$f \in [\![\mathbb{N} \to \mathbb{N}]\!] \mid x \mapsto x!$ (la factorielle de x)

sera atteinte comme la limite de la chaîne non stationnaire des

$f_i \in [\![\mathbb{N} \to \mathbb{N}]\!] \mid x \mapsto$ si $x \le i$ alors $x!$ sinon $\perp$.

Par contre, f(x) sera atteint comme le dernier élément de la chaîne stationnaire des $f_i(x)$.

La fonction factorielle des mathématiciens est définie sur un ensemble infini de valeurs (tout $x \in \mathbb{N}$) ; or, en informatique, s'il est bien nécessaire de définir la fonction factorielle pour tout nombre entier, la valeur de la fonction factorielle ne sera jamais demandée que pour une valeur de $\mathbb{N}$ bien concrète.

Donc, la fonction factorielle définie pour tout entier est la limite d'une suite infinie de fonctions ; alors que la fonction factorielle nécessaire lors d'un appel concret est la plus grande valeur d'une suite finie de fonctions : la chaîne des fonctions qui approximent la valeur demandée.

Il nous faut donc :

1. faire en sorte que l'ensemble des fonctions $A \to B$ calculant un problème particulier soit un espace inductif.
2. faire en sorte que l'ensemble des fonctions $A \to B$ calculant un problème particulier pour une valeur particulière soit un espace strictement inductif.

Pour que le domaine des fonctions soit inductif, il faut

- une relation d'ordre (ce qu'on a déjà)
- un lub pour toute chaîne croissante
  ce qui sera la propriété de continuité des fonctions.

Notion de domaine fonctionnel

Nous avons déjà ordonné les fonctions par

$f \in [\![A^+ \to R^+]\!] \mid x \mapsto f(x)$

$g \in [\![A^+ \to R^+]\!] \mid x \mapsto g(x)$

$f \preceq g \Leftrightarrow \forall x \in A^+ \; f(x)=g(x) \vee f(x)=\perp$

Mais cette définition de $\preceq$ était limitée aux domaines primitifs.

La définition suivante étend $\preceq$

$(f,g \in [\![A \to B]\!]) \Rightarrow (f \preceq g \Leftrightarrow \forall x \in A \; f(x) \preceq g(x))$

Appelons $[\![A \to B]\!]$ avec A et B domaines inductifs, un domaine fonctionnel.
(rappel : on ne considère que les fonctions monotones)

Tout domaine fonctionnel $[\![A \to B]\!]$ a un plus petit élément : la fonction nulle part définie :

$\exists \perp_{[A \to B]} \in [\![A \to B]\!] \mid \forall a \in A \; \perp_{[A \to B]}(a) = \perp_B$

$\wedge \; \forall x \in [\![A \to B]\!] \; \perp_{[A \to B]} \preceq x$

On peut aisément démontrer que l'ensemble des fonctions de A dans B avec A et B espaces inductifs est inductif.

Soit une chaîne de fonctions $\langle f_i \rangle = f_0 \preceq f_1 \preceq f_2 \preceq f_3 \preceq f_4 \preceq \ldots$ dans $[\![A \to B]\!]$
avec $\preceq$ défini par : $\forall f,g \in [\![A \to B]\!] \; f \preceq g \Leftrightarrow \forall a \in A \; f(a) \preceq g(a)$.

Démontrons que cette chaîne a une plus petite borne supérieure dans $[\![A \to B]\!]$.

Rappelons la définition :

$lub\langle f_i \rangle = f \Leftrightarrow (\forall i \geq 0 \; f_i \preceq f) \wedge (\forall g \in [\![A \to B]\!] \; (\forall i \geq 0 \; f_i \preceq g) \Rightarrow f \preceq g)$

Soit a un élément quelconque de A, on peut construire la chaîne

$f_0(a) \preceq f_1(a) \preceq f_2(a) \preceq f_3(a) \preceq f_4(a) \preceq \ldots$

comme tous ces éléments appartiennent à B et que B est inductif, cette chaîne a un lub : $lub\langle f_i(a) \rangle$
Ce raisonnement peut être tenu pour tous les éléments de A : à chaque élément de A correspondra une valeur $lub\langle f_i(a) \rangle$
Définissons la fonction $f \in [\![A \to B]\!] \mid \forall a \in A \; f(a) = lub\langle f_i(a) \rangle$
On a bien que $(\forall i \geq 0 \; f_i \preceq f)$
puisque $(\forall i \geq 0 \; \forall a \in A \; f_i(a) \preceq lub\langle f_i(a) \rangle = f(a))$
Est-ce que $(\forall g \in [\![A \to B]\!] \; (\forall i \geq 0 \; f_i \preceq g) \Rightarrow f \preceq g)$?
Oui, sinon il existerait a tel que $g(a) \prec f(a) = lub(f_i(a))$
et donc qu'il existe i tel que $g(a) \prec f_i(a)$
ce qui exclut que $f_i \preceq g$

Une chaîne de fonctions (monotones) dans un domaine fonctionnel définit une fonction ; inversement, une fonction peut être définie par une chaîne.

Notion de continuité

Définir une fonction à partir d'une chaîne de fonctions voilà qui est intéressant confer plus haut la notion de convergence, d'approche d'une fonction par approximations successives. Mais il peut y avoir des surprises :

Soit la fonction $f \in [\![[0..2] \to [0..2]]\!]$: $x \mapsto$ si $x<1$ alors 0 sinon 1
avec [0..2] l'ensemble des nombres réels compris entre 0 et 2 inclus.
$([0..2],\leq)$ est bien un espace inductif et f est monotone.

On peut construire la chaîne infinie des nombres réels de [0..1[(le nombre 1 exclus) ; et l'on aura

$lub(x_i)=1 \quad f(lub(x_i))=1 \neq lub(f(x_i))=0$

en effet chaque $f(x_i)$ de cette chaîne vaut 0 et donc $lub(f(x_i))=0$.

Pour que nous puissions définir une fonction (de table infinie) comme la limite d'une suite de fonctions (de table finie) on va devoir exiger que nos fonctions soient continues, c'est-à-dire :

$f \in [\![A \to B]\!]$ avec A et B inductifs est continue
ssi pour toute chaîne de $x_i \in A$
$f(lub\langle x_i\rangle)=lub(f(\langle x_i\rangle))$

On dit aussi que f est continue ssi elle préserve les limites ; en effet $lub\langle x_i\rangle$ est la limite des x_i alors que $lub(f(\langle x_i\rangle))$ est la limite des $f(\langle x_i\rangle)$.

On peut remarquer qu'une fonction continue est forcément monotone : si l'on a $x\preceq y\preceq y\preceq\ldots$ la borne supérieure de cette chaîne est y et donc $lub\langle x\preceq y\preceq y\preceq\ldots\rangle=y$ donc $f(lub\langle x\preceq y\preceq y\preceq\ldots\rangle)=f(y)$ et $f(x)\preceq f(y)$.

....

Nous avons déjà établi que A*B*C doit être lu (A*B)*C : associativité à droite.
Ajoutons maintenant que l'opérateur * est prioritaire face à →.
A*B→C→D*E doit être lu (A*B)→C→(D*E).
Cependant f ∈ A*B→C*D pourrait être considéré comme une fonction qui reçoit un paramètre de type A et retourne une fonction qui demande un paramètre de type B puis retourne un produit cartésien de type C*D. Ce procédé est intéressant dans la mesure où il permet d'éliminer les parenthèses. On n'écrira plus f(a,b) mais fab qui doit être lu comme équivalent à (fa)b.
On retrouve ici l'associativité à droite mais il faut prendre garde : comment lire $f \in [\![A\to B\to C\to D]\!]$?
Si l'on dit que fabcd = (((fa)b)c)d
f est une fonction qui reçoit un argument de type A et retourne une fonction de type B→C→D. Donc il faut lire $f \in [\![A\to[\![B\to[\![C\to D]\!]]\!]]\!]$.
L'associativité est ici à gauche !!!

Notion de transformation

Appelons transformation toute fonction d'un ensemble dans lui-même.

Posons $\tau \in [\![A \to A]\!]$ avec A inductif

Considérons que τ est monotone : $\forall f,g\in A \; f\preceq g \Rightarrow \tau(f)\preceq\tau(g)$

Soit $\langle f_i\rangle$ une chaîne dans A : $f_0\preceq f_1\preceq f_2\preceq f_3\preceq f_4\preceq\ldots$

Comme τ est monotone, on peut écrire :
$\tau(f_0)\preceq\tau(f_1)\preceq\tau(f_2)\preceq\tau(f_3)\preceq\tau(f_4)\preceq\ldots$

Cette chaîne est dans A qui est inductif ; cette chaîne a donc une plus petite borne supérieure qui est $lub(\tau\langle f_i\rangle)$.

si τ est continue on aura $\forall\langle f_i\rangle \; \tau(lub\langle f_i\rangle)=lub(\tau\langle f_i\rangle)$

Construisons maintenant la suite d'éléments de A définie par

$f_0=\bot$
$f_1=\tau(f_0)$
$f_2=\tau(f_1)$
...
$f_{i+1}=\tau(f_i)$
...

On a que $f_0=\bot\preceq f_1$
Comme τ est monotone, on a que $\tau(f_0)\preceq\tau(f_1)$ d'où $f_1\preceq f_2$
Puis $\tau(f_1)\preceq\tau(f_2)$ d'où $f_2\preceq f_3$
et ainsi de suite, pour arriver à :

$$\bot \preceq \tau(\bot) \preceq \tau(\tau(\bot)) \preceq \tau(\tau(\tau(\bot))) \preceq \ldots \preceq \tau^n(\bot) \preceq \ldots$$

Comme τ est continue, on a que : $\tau(\mathrm{lub}\langle\tau^i(\bot)\rangle) = \mathrm{lub}(\tau(\langle\tau^i(\bot)\rangle))$

Or qu'est-ce que la chaîne $\tau(\langle\tau^i(\bot)\rangle)$ sinon

$$\tau(\bot) \preceq \tau(\tau(\bot)) \preceq \tau(\tau(\tau(\bot))) \preceq \ldots \preceq \tau^n(\bot) \preceq \ldots$$

dont la borne supérieure ne peut être que la même que

$$\bot \preceq \tau(\bot) \preceq \tau(\tau(\bot)) \preceq \tau(\tau(\tau(\bot))) \preceq \ldots \preceq \tau^n(\bot) \preceq \ldots$$

Ce qui nous donne $\tau(\mathrm{lub}\langle\tau^i(\bot)\rangle) = \mathrm{lub}(\tau(\langle\tau^i(\bot)\rangle)) = \mathrm{lub}\langle\tau^i(\bot)\rangle$

Si on désigne $\mathrm{lub}\langle\tau^i(\bot)\rangle$ par x, on a $\tau(x)=x$.

On en conclut que à toute transformation continue d'un ensemble inductif dans lui-même correspond un unique élément $\mathrm{lub}\langle\tau^i(\bot)\rangle$; cet élément peut être construit par les approximations successives que sont

$$\bot \preceq \tau(\bot) \preceq \tau(\tau(\bot)) \preceq \tau(\tau(\tau(\bot))) \preceq \ldots \preceq \tau^n(\bot) \preceq \ldots$$

Comme $\tau(x)=x$, x est un point fixe de la transformation τ.

Une transformation peut avoir plusieurs points fixes ; en effet si $\tau(x)=x$ et $x\prec y$ on aura $\tau(y)=y$ et y est un point fixe.

Mais l'unique élément $\mathrm{lub}\langle\tau^i(\bot)\rangle$ est le plus petit point fixe de τ ; cet élément existe toujours.

En effet soit y tel que $\tau(y)=y$; on sait que $\bot\preceq y$ et donc comme τ est monotone $\tau(\bot)\preceq\tau(y)$ d'où $\tau(\bot)\preceq y$; si on applique encore une fois la monotonie, on trouve $\tau(\tau(\bot))\preceq\tau(y)$ d'où $\tau(\tau(\bot))\preceq y$ et ainsi de suite jusqu'à $\mathrm{lub}\langle\tau^i(\bot)\rangle\preceq y$.

Notion de fonctionnelle

On vient de démontrer qu'à toute transformation τ au sein d'un ensemble inductif, correspond un et un seul élément x de cet ensemble tel que $\tau(x)=x$ et qui est défini par $x=\mathrm{lub}\langle\tau^i(\bot)\rangle$.

Si nous prenons comme ensemble inductif de départ l'ensemble des fonctions monotones, on constate que f est une fonction totale alors que chaque élément de la suite des $\langle\tau^i(\bot)\rangle$ est une fonction partielle : $\bot$ n'est défini nulle part puis $\tau(\bot)$ est un peu mieux définie ($\bot \preceq \tau(\bot)$) puis on avance ainsi pas à pas jusqu'à la limite qu'est $f=\mathrm{lub}\langle\tau^i(\bot)\rangle$. Autrement dit, la fonction totale f est accessible comme la limite d'une suite de fonctions partielles de mieux en mieux définies.

Si l'on regarde maintenant le problème à l'envers, un élément f de l'ensemble inductif des fonctions monotones peut être défini à partir d'une transformation continue de fonctions τ telle que $\tau(f)=f$.

Appelons <u>fonctionnelle</u> toute transformation dans le domaine des fonctions.

Une fonctionnelle (continue) τ permet de <u>construire</u> une fonction f par approximations successives comme la plus petite borne supérieure de

$$\perp \preceq \tau(\perp) \preceq \tau(\tau(\perp)) \preceq \tau(\tau(\tau(\perp))) \preceq \ldots \preceq \tau^n(\perp) \preceq \ldots$$

Cette suite peut être non stationnaire (se composer d'un nombre infini d'éléments distincts) mais là n'est pas le problème !

Pour exprimer que f peut être ainsi construit on dira que $f=\mu(\tau)$.

On appellera $\mu(\tau)$, <u>le point fixe</u> de la transformation τ.

Calcul d'une fonction par point fixe

Rappelons-nous la définition de la convergence :

$\langle f_i \rangle \subseteq [A \to B]$
$f \in [A \to B]$
$\langle f_i \rangle$ converge vers f ssi
$\forall i \geq 0 \; f_i \preceq f_{i+1} \wedge \forall a \in A \; \exists i \geq 0 \mid f_i(a)=f(a)$

Or si l'on découvre $\tau \in [\, [A \to B] \to [A \to B] \,]$ et tel que $f=\mu(\tau)$, on a construit un moyen de calculer f et donc f(x). Or seul f(x) nous intéresse, donc pour connaître f(x) : il suffit de construire la suite des $\langle f_i \rangle$ c'est-à-dire des $\langle \tau^n \rangle$ jusqu'à ce que f_i est définie pour x.

D'après la définiton du point fixe,
si $f=\mu(\tau)$ alors $f(x)=(\text{lub}\langle \tau^i(\perp) \rangle)(x)$
et donc $\exists i \geq 0 \mid \tau^i(\perp)(x)=f(x)$

i sera-t-il grand ou petit ? c'est tout le problème de la convergence. Comment définir τ ? c'est tout le problème des définitions récursives.

Notion de fonction récursive

L'écriture <u>$f=\tau(f)$</u> est récursive puisque f est l'inconnue et que f se trouve à droite comme à gauche du symbole =.

Or la théorie qui vient d'être exposée nous fournit une méthode de calcul de f (dans la définition de τ ci-dessus et de son point fixe, f n'apparaît pas et pourtant τ définit f) et donc une méthode universelle de résolution des équations récursives. Mieux encore : la théorie du point fixe énonce qu'il existe une et une seule plus petite solution.

Toute fonction récursive peut maintenant être définie comme le plus petit point fixe d'une transformation continue et être calculée par approximations successives.

Remarque : de la même manière que l'on a pu dire qu'une constante ce n'est rien d'autre que l'application d'une fonction constante, une fonction non récursive n'est rien d'autre qu'une transformation constante. Au contraire de ce qu'on pourrait croire les fonctions non récursives sont des cas particuliers...

Supposons que nous ayons à définir les opérations sur $\mathbb{N}$ et que la seule primitive disponible soit
$s \in [\mathbb{N}^+ \to \mathbb{N}^+] \mid x \mapsto x+1$

On peut définir la fonction précédent comme

```
précédent ∈ ⟦N⁺→N⁺⟧ | x ↦ préc(x,0)
préc ∈ ⟦N⁺*N⁺→N⁺⟧ |
          (x,y) ↦ si x=0 alors ⊥
                         sinon si x=1 alors y
                                      sinon préc(s(x),s(y))
```

Cette dernière fonction est récursive ; pour exposer sa <u>SÉMANTIQUE</u>, nous dirons dorénavant que préc est l'unique solution de

```
μτ (le plus petit point fixe de τ)
avec τ définit par
τ ∈ ⟦N⁺*N⁺→N⁺*N⁺⟧ :
     (x,y) ↦ si x=0 alors ⊥
                  sinon si x=1 alors (x,y)
                               sinon (s(x),s(y))
```

Si l'on définit la fonction + par

```
+ ∈ ⟦N⁺*N⁺→N⁺⟧ | (x,y) ↦ si x=0 alors y sinon +(préc(x),y)
```

On devra désormais considérer que l'on énonce en fait :

+ est le plus petit point fixe de la transformation τ_1

τ_1

Remarquons qu'une définition récursive n'a pas toujours un point fixe : f(x) = f(x)+1

Encore un exemple :
La fonction définie par la fonctionnelle

```
τ ≬ [λf.λx.if x=0 then 1 else x*f(x-1)]
```

peut être calculée en remplacant f par ⊥ puis τ(⊥), puis τ(τ(⊥)), etc ce qui donne

```
τ⊥ = [λx.if x=0 then 1 else x*⊥(x-1)]
   = [λx.if x=0 then 1 else ⊥]
τ(τ⊥) = [λf.λx.if x=0 then 1 else x*f(x-1)][λx.if x=0 then 1 else ⊥]
      = [λx.if x=0 then 1 else x*[λx.if x=0 then 1 else ⊥](x-1)]
      = [λx.if x=0 then 1 else x*[if (x-1)=0 then 1 else ⊥]]
      = [λx.if x=0 then 1 else if (x-1)=0 then 1 else ⊥]
τ(τ(τ⊥)) = [λf.λx.if x=0 then 1 else x*f(x-1)][λx.if x=0 then 1 else if (x-1)=0 then 1 else ⊥]
         = [λx.if x=0 then 1 else x*[λx.if x=0 then 1 else if (x-1)=0 then 1 else ⊥](x-1)]
         = [λx.if x=0 then 1 else x*[if (x-1)=0 then 1 else if ((x-1)-1)=0 then 1 else ⊥]]
         = [λx.if x=0 then 1 else if x=1 then 1 else if x=2 then 2 else ⊥]
```

On constitue ainsi une suite infinie de fonctions de plus en plus définies.

$\perp \preceq \tau(\perp) \preceq \tau(\tau(\perp)) \preceq \tau(\tau(\tau(\perp))) \preceq \ldots$

On arrêtera dans la substitution dès que l'on a atteint la fonction qui est définie pour la valeur du paramètre.

L'écriture traditionnelle :

```
f(x) = if x=0 then 1 else x*f(x-1)
```

signifie que l'on ne peut demander la valeur de f (c'est une table infinie) mais bien de f(x).
Comme f est sémantiquement défini comme le plus petit point fixe de

```
τ ≬ [λf.λx.if x=0 then 1 else x*f(x-1)]
```

si l'on désigne par μ l'opérateur de point fixe, $\mu(\tau)$ défini la fonction f mais est infini alors que $\mu(\tau)$x calcule la valeur de la fonction f pour x <u>comme la limite d'une suite de fonctions partielles</u>.

Récursivité croisée

Jusqu'ici, nous avons soigneusement évité de considérer des définitions récursives du type de :

f(x) = ... g(y) ...
g(a) = ... f(b) ...

Pour résoudre ce problème - d'une manière assez lapidaire il est vrai - , il nous suffira de considérer que l'on a ici non pas deux fonctions mais une seule, la fonction fg(x,a). La démonstration en est fournie dans La fiabilité des programmes de H. Leroy. Nous considérerons ainsi que l'appel f(x) "c'est" l'appel fg(x,⊥) et que quand lors de cette exécution de f pour x il est fait appel à g(a) "c'est" l'appel à fg(x,a).

Notion de treillis

Cette notion de treillis n'est nullement nécessaire à la théorie du point fixe et notre programme aurait pu s'en passer : nos domaines auraient pu être simplement inductifs. La notion de domaine inductif a d'abord été introduite pour donner un sens au "plantage" d'un programme sur une donnée débordant de son domaine de validité et pour exprimer qu'une fonction informatique doit être monotone, c'est à dire qu'en comparant deux exécutions, si la seconde exécution reçoit des données "moins définies" que la première, elle doit fournir un résultat "moins défini" : $x \preceq y \rightarrow f(x) \preceq f(y)$.

Il est une autre classe de situations qui peut poser problème ; en voici quelques exemples : division par zéro, débordement d'indice, incompatibilité de types, lub de x et y tel que $x \npreceq y$ et $x \nsucceq y$, ...

On pourrait donner un sens sémantique à ces situations comme on l'a fait avec ⊥ pour "donnée non initialisée" en ajoutant un nouvel élément, cette fois appelé ⊤, et qui signifierait "surdéfinition". Tout comme ⊥ est "le plus petit", on fera en sorte que ⊤ est "le plus grand". L'avantage de ⊤ est que - tout comme avec ⊥ - l'on pourra assurer la totalité et la monotonie des fonctions.

(T,≺) ensemble (partiellement) ordonné est un treillis ssi

$\forall a,b \in T\ \exists s \in T \mid a \preceq s \wedge b \preceq s \wedge (\forall s' \in T\ ((a \preceq s' \wedge b \preceq s') \Rightarrow s \preceq s'))$
$\forall a,b \in T\ \exists i \in T \mid i \preceq a \wedge i \preceq b \wedge (\forall i' \in T\ ((i' \preceq a \wedge i' \preceq b) \Rightarrow i' \preceq i))$

(T,≤) est ainsi muni de deux fonctions binaires totales :

$\square : T*T \rightarrow T : (a,b) \mapsto a \square b$
$\mid a \preceq a \square b \wedge b \preceq a \square b \wedge (\forall s' \in T\ ((a \preceq s' \wedge b \preceq s)' \vee a \square b \preceq s'))$
a□b est dit le "supremum" ou "borne supérieure" ou "lub"

$\Diamond : T*T \rightarrow T : (a,b) \mapsto a \Diamond b$
$\mid a \Diamond b \preceq a \wedge a \Diamond b \preceq b \wedge (\forall i' \in T\ ((i' \preceq a \wedge i' \preceq b) \Rightarrow i' \preceq a \Diamond b))$
a◊b est dit l'"infinum" ou "borne inférieure" ou "glb"

C'est le choix de la loi d'ordre qui définira ⊥ et ⊤. On désignera un treillis par le doublet (T,≤). Plus précisément :

T est un treillis <->

$\exists \preceq$ relation d'ordre partiel dans T

c'est-à-dire . $\preceq : T*T \rightarrow \{VRAI,FAUX\} : (a,b) \mapsto a \preceq b$

. $\forall a \in T \; a \preceq a$

. $\forall a,b \in T \; (a \preceq b \wedge b \preceq c) \Rightarrow a \preceq c$

. $\forall a,b \in T \; (a \preceq b \wedge b \preceq a) \Leftrightarrow a=b$

$\wedge \exists \bot \in T \mid \forall x \in T \Rightarrow \bot \preceq x$

$\wedge \exists \top \in T \mid \forall x \in T \Rightarrow x \preceq \top$

$\wedge \exists \square$ opérateur total "lub"

défini par $\square : T*T \rightarrow T : (a,b) \mapsto a \square b$

$\mid (a=b \square c) \Leftrightarrow b \preceq a \wedge c \preceq a \wedge$

$(\nexists x \in T \mid b \preceq x \wedge c \preceq x \wedge x \preceq a \wedge x \neq a)$

$\wedge \exists \lozenge$ opérateur total "glb"

défini par $\lozenge : T*T \rightarrow T : (a,b) \mapsto a \lozenge b$

$\mid (a=b \lozenge c) \Leftrightarrow a \preceq b \wedge a \preceq c \wedge$

$(\nexists x \in T \mid x \preceq b \wedge x \preceq c \wedge a \preceq x \wedge a \neq x)$

Dans un treillis, quelque soit a et b, il existe toujours $a \square b$ et $a \lozenge b$: deux éléments ne sont pas toujours comparables (ni $a \preceq b$, ni $b \preceq a$), mais il existe toujours un plus petit élément et un plus grand élément par rapport auquels ils sont comparables ($a \preceq s$ et $b \preceq s$; $i \preceq a$ et $i \preceq b$) ; de plus, parmi tous les éléments ainsi définis ("majorants" et "minorants") il y a un qui est respectivement le plus petit ("supremum"), le plus grand ("infinum").

Remarquons qu'un treillis peut être un ensemble infini et que dans un treillis tous les "plus grands que a et b" doivent être comparables entre eux : se trouvent sur un même chemin menant à $\top$; de même pour les "plus petits que a et b".

Notion de sémantique dénotationnelle

La sémantique dénotationnelle expose la signification d'un programme informatique en définissant une fonction : la fonction des inputs vers les outputs. A cette fonction correspond une table, généralement infinie, mais mathématiquement définie.

La sémantique dénotationnelle du programme

```
integer inc(integer x)
{ return x+1 ;
}
```

est tout simplement $(inc)x \triangleq x+1$
mais le + ici mentionné n'est pas le même + que plus haut !
il s'agit du + du métalangage supposé bien défini.

La sémantique dénotationnelle est indépendante de l'algorithme effectivement utilisé.

La sémantique dénotationnelle du programme

```
integer inc(integer y)
{ Bigfloat z=factorielle(8427) ;
  return y+1-1+2-2+1 ;
}
```

est également $(inc)x \triangleq x+1$
si toutefois factorielle(8427) a une solution ...
tout simplement parce que $x+1 \equiv x+123456789-123456788$
en effet, ces deux fonctions donnent exactement la même table.

En sémantique dénotationnelle, on énonce la table de correspondance qu'un programme dénote. L'exécution d'un programme n'est qu'une lecture (instantanée) de cette table. La sémantique dénotationnelle d'un programme est l'exposé d'une fonction mathématique dont la table des correspondances est identique à la table obtenue par toutes les éxécutions possibles du programme dans le domaines des valeurs d'input.

Quand la table est infinie, pour définir chacune des valeurs possibles, la sémantique dénotationnelle exploite le μ-calcul, c'est-à-dire définit la fonction comme le plus petit point fixe d'une transformation continue.

Le programme suivant :

```
     unsigned len(char *t)
     { unsigned p=0 ; while (*p++) n++ ;
     }
```

a pour sémantique : μPt0
avec P une transformation définie par

```
     P : unsigned x char* -> unsigned x char*
          Ptn ≜ if (*t) then (t--)(n++) else tn
```

où toute récursivité a disparue.