2.2 Capacité et identité personnelles

2.2.1 Le soi et l'univers

L'ensemble des fonctions partielles calculables est récursivement énumérable. Celles-ci seront toujours désignées par une lettre ϕ avec un index représentant le numéro de code dans l'énumération, ou encore par un abus que je justifierai avec l'illustration en LISP, le code lui-même :

$$P = \{\phi_0, \phi_1, \phi_2, \phi_3, ...\},$$

ϕ_i, représente donc l'*extension*, ou encore le comportement extensionnel, c-à-d l'ensemble des entrées-sorties du programme i. Mathématiquement, c'est le graphe de la fonction ϕ_i de ω dans ω, calculée par le $i^{ème}$ code. Ce code dénote, sans ambiguïté ϕ_i relativement à un langage universel choisi pourvu qu'il calcule les fonctions lambda-définissables.

i représente l'*intension*, (avec s), i est appelé index, code, programme, ou machine, pour la fonction ϕ_i, relativement au langage L; (i x) représente ce que j'appelle une forme. Il s'agit de la description du couple constitué d'un code (par exemple un nombre naturel, mais dans les illustrations il s'agira d'une lambda-expression) et d'une description des arguments. La suite des états du calcul évaluant la forme (i x) est ce que j'appelle le comportement intensionnel. On l'obtient par itération de la dynamique E-step.

Définition: R est l'ensemble de toutes les fonctions totales calculables, P est l'ensemble (énuméré) de toutes les fonctions partielles calculables.

Définition Une numérotation P des fonctions partielles calculables est dite *acceptable* si P satisfait, comme les énumérations de l'école du dehors, aux deux propriétés fondamentales (Rogers 1958) : l'existence d'une machine universelle, et l'existence d'une machine paramétrisatrice.

Remarques

a) On pourrait donc travailler axiomatiquement, mais on gardera à l'esprit les énumérations standards possibles. Pour chacune d'entre elle ces deux axiomes sont évidemment des théorèmes.

b) Je mentionne néanmoins, et utiliserai accessoirement un autre modèle de numérotation acceptable comme l'énumération des machines de Turing (ou des programmes LISP, des fonctions lambda-définissables, etc.) ayant à leur disposition un oracle (Turing 1936). Un oracle est simplement un réel dont la machine peut consulter des parties finies en cours d'éxécution. On peut le regarder comme un être divin auquel la machine peut de temps à autre poser une question. Un oracle important, que j'appelle *le réel de Post-Turing* et que je note ρ, est donné par la solution au problème de l'arrêt, codée par un réel binaire : $0,I_1I_2I_3$ ..., où $I_j = 1$ ou 0 selon que $\phi_i(x)$ est définie ou non. ρ désigne en fait un schéma de tels réels dépendant du choix de la machine universelle sous-jacente.

Notons que la procédure qui consiste à observer les machines suffisamment longtemps, en partant de 0,0000..., et en corrigeant les décimales chaque fois qu'une machine s'arrête, se stabilise, de façon non prouvable, dans les voisinages de l'infini. On peut justifier cela en utilisant le *busy beaver* BB (voir plus loin). Voir Soare 1980 pour la démonstration d'un résultat apparenté dû à Shoenfield.

Notons encore que les énumérations des machines avec oracles sont fermées pour la diagonalisation, ce qui permet de prouver l'existence de problèmes insolubles. Les oracles ont permis l'étude des *degrés d'insolubilité,* (Turing 1936, Post 1944) qui constitue l'essentiel de la théorie de la récursion.

1°) l'existence d'une machine universelle :

$$\exists u \ \forall i \forall x \quad \phi_u(i,x) = \phi_i(x) \qquad (1)$$

C'est-à-dire, il existe une machine, *u* désigne son index, capable de calculer chaque ϕ_i, si on lui donne comme argument l'index *i* et l'argument x. L'existence explicite de telle machine est un résultat dû à Turing 1936 dans

le domaine des fondements des mathématiques, et son importance concernant l'ensemble des fonctions partielles calculables P, (identifié ici par abus avec les numérotations acceptables) justifie, je le rappelle (voir 2.1) l'affirmation selon laquelle l'ordinateur a été (re)découvert par les métamathématiciens dans les années 30.

Definition: J'appellerai *u*, qui est finiment descriptible, un univers singulier, ou un environnement universel ou un niveau d'universalité, ou encore, dans certains cas, un univers. Remarquons que l'univers des physiciens (le cosmos), le cerveau, le système génétique, mais aussi les individus humains, sont ***au moins*** des univers, les ordinateurs sont, par construction -ou par utilisation- au plus des univers. Et le mécanisme digital indexical énonce que nous sommes au plus des environnement universels. C'est parce que les valeurs sont relatives à un environnement que j'identifie la machine universelle avec un environnement. Celui-ci pourra comporter des données infinies (comme des oracles) sans que cela ne change rien à la plupart des démonstrations.

2°) l'existence d'une machine paramétrisatrice

Definition :
Une machine qui admet comme sortie des machines (ou des nombres considérés comme des descriptions de machines) sont appelées métaprogrammes (métamachines ou encore des opérateurs). Elles sont elles-mêmes énumérables par

$$\phi\phi_i$$

Deux exemples :

1) les suites de machines $\phi\phi_i(j)$.

2) le composeur

$$\exists h \forall i \forall j \quad \phi_{\phi_h(i,j)} = \phi_i \circ \phi_j$$

h est le code d'une fonction qui, appliquée aux codes i et j donne le code d'une fonction calculant la composition de ϕ_i et ϕ_j.

La deuxième propriété des énumérations acceptables est épinglée par le théorème d'existence d'un paramétrisateur qui vaut pour les énumérations ambitieuses, mais aussi pour les énumérations assez riches des écoles du dedans :

Théorème de la paramétrisation (pour les énumérations standards) :

$$\exists s \ \forall i \ \forall x \forall y \ \phi_i(x,y) = \phi_{\phi_S(i,x)}(y). \qquad (2)$$

s est un métaprogramme qui paramétrise une fonction de n+1 variables[1] en une fonction de n variables en fixant un de ses arguments. Par exemple :

$$\phi_S((\lambda x \lambda y \ x+y), 2) = \lambda y \ 2+y.$$

Voici *s* écrit en LISP[2] (voir annexe 1 pour plus de détails, et pour le code des sous-routines) :

```
(def 's '(lambda (programme donnée)
  (liste (quote lambda)
      (enlève-premiers (longueur donnée)
                       (arguments programme))
      (substitue-arguments (mapquote donnée)
                  (premiers (longueur d)(arguments
                                         programme))
                  (corps programme)))))
```

s s'appelle le programme paramétrisateur, ou la machine paramétrisatrice.

Substitue-arguments appelle une routine subst-sauf-quote qui est une routine de substitution de liste, y compris à l'intérieur des sous-listes. C'est elle qui permet l'identification pratique, dans Φ-LISP, entre programme et liste, de même qu'on va identifier dans les contextes de duplication de soi nombre et machine.

Exemple : (s '(lambda (x y z) (F (G y z))) '(3 4)) donne (LAMBDA (Z) (F (G '4 Z))). Une simple fonction de substitution n'aurait pas introduit un quote devant le nombre 4, substitue-arguments introduit un quote de façon à gérer localement l'identification entre nombres et programmes.

Remarque : le théorème de la paramétrisation marche pour les école du dedans.

---

[1] A strictement parler, les fonctions de n variables f(x, y, ...z) = f(<x,y,...z>) sont des fonction à un argument (une entrée). On se souvient que j'ai décidé (voir 2.1.) de commettre cet abus de langage.

[2] En pure Lisp, où SCHEME, où en calcul λ, ce serait plus facile, mais la façon dont je procède fonctionne pour (plus que) les numérotations acceptables. En pure lisp SMN =... . On ne distingue plus la récursion intensionnelle de l'extensionnelle. L'approche ici est plus générale.

2.2.2 Capacité personnelle

Je propose ici une définition de l'identité personnelle et de la capacité personnelle d'une machine relativement à l'environnement (machine universelle avec ou sans oracles). La définition permettra de modéliser (conceptuellement) la duplication, la reproduction biologique (la solution est compatible avec la solution mise en évidence par les généticiens moléculaires chez les organismes biologiques), mais permettra aussi une interprétation mécaniste digitale de la théorie de la conscience présentée en UN. Cette théorie sera étendue dans la troisième partie en une théorie (informelle) du rêve et de la réalité. Nous verrons que le mécanisme digital, lorsqu'il devient indexical collapse en quelque sorte (et sur un niveau nécessairement insaisissable) la notion de modèle et la réalité, ce qui sera utilisé pour entreprendre une formulation du problème du corps et de l'esprit en philosophie mécaniste.

La définition proposée pour la capacité personnelle repose intégralement sur l'énoncé et la démonstration du *théorème de Kleene*, connu comme le second théorème de récursion 2-REC :

$$\forall t \quad \exists e \quad \forall y \qquad \phi_e(y) = \phi_t(e,y) \qquad\qquad (2\text{-REC})$$

En d'autres mots, pour toute transformation *t*, incarnée (représentée) par la machine <code, index, nombre, mot> il existe une machine *e* qui, sur n'importe quelle entrée *y*, calcule *t* appliquée sur sa propre représentation (incarnation) accompagnée de la donnée *y*. Remarquons que les quantificateurs portent exclusivement sur des nombres naturels. 2-REC est une proposition arithmétique (ou un schéma de telles propositions selon la machine universelle choisie au départ).

Preuve

L'expression $\lambda x \lambda y.\phi_t(\phi_s(x,x),y)$, où t est le code de la transformation et s le code du paramétrisateur (première diagonalisation) désigne une fonction intuitivement calculable. Donc, il existe une machine r telle que ϕ_r calcule cette expression :

$$\phi_r(x, y) = \phi_t(\phi_S(x, x), y) \qquad \text{(1ère diagonalisation)}$$

En utilisant le théorème de la paramétrisation sur le côté gauche de cette équation :

$$\phi_r(x,y) = \phi_{\phi_s(r,x)}(y)$$

Par instantiation universelle, soit x = r, c'est la deuxième diagonalisation :

$$\phi_t(\phi_s(r, r), y) = \phi_{\phi_s(r, r)}(y) \qquad \text{(2ème diagonalisation)}$$

On a terminé en prenant $e = \phi_s(r,r)$. QED.

Voici une fonction de diagonalisation écrite en Φ-LISP :

```
(def 'diag '(lambda (f)
  (liste (quote lambda)
       (arguments f)
       (substitue-arguments
        (liste (quote s)
             (car (arguments f))
             (list (quote list)
                  (car (arguments f))))
        (car (arguments f))
        (corps f)
       ))
))
```

exemple : (diag '(lambda (x) (length x))) donne

(lambda (x) (longueur (s x (liste x))))

La double diagonalisation est capturée de façon uniforme par le programme suivant :

```
(def 'k '(lambda (f)
  (s (diag f) (liste (diag f))))
```

Ainsi k appliquée au code de la transformation t va construire une machine e qui applique t sur elle-même (au niveau de la représentation relative à l'environnement universel).

Définition J'appelle k le *métaprogramme, ou encore l'opérateur de Kleene*, parce qu'il illustre l'uniformisation de la preuve du théorème de Kleene.

exemple :

(k '(lambda (x) (longueur x))) donne

```
(lambda nil
     (longueur (s '(lambda (x) (length (s x (list x))))
          (list '(lambda (x) (length (s x (list x))))))))
```

Ce programme, utilisé sans données, donne sa propre longueur.

Definition Une machine *M* a une capacité personnelle par rapport à une transformation *t* relativement à un environnement universelle *u* si M = *(k t)* interprétée par *u*.

Cela signifie que *M* est capable d'appliquer *t* sur sa propre représentation dans u (et au moyen de u). La capacité personnelle, comme elle est présentée ici est une notion relative et intensionnelle.

2.2.3 L'*amibe*, l'identité personnelle et le problème de Descartes

Une machine capable de se reproduire relativement à un environnement, comme l'amibe ou la personne, candidate à la pratique du mécanisme, qui construit son propre double, est une machine qui a une capacité personnelle par rapport à la fonction identité. Une telle machine est capable d'appliquer la fonction identité, I = λx x, sur elle-même :

$$\phi_e() = I(e) = (\lambda x\ x)(e) = e$$

exemple: si l'environnement universel est LISP, alors I = (lambda (x) x) :

```
(def 'I '(lambda (x) x) )          (en Φ-LISP)
```

Donc, pour obtenir une *amibe* -une machine capable de se reproduire- et résoudre ainsi le problème de Descartes, il suffit d'appliquer le métaprogramme *k* de Kleene sur une représentation dans LISP, de la fonction identité :

$$\phi_k(I) = (k\ I) =$$

```
(lambda nil
  (s '(lambda (x) (s x (list x)))
     (list '(lambda (x) (s x (list x))))))
```

Appliquer ce programme sur l'entrée vide donne lui-même:

$$\phi_{\phi_k(I)}() = ((k\ I))$$

donne

```
(lambda nil
  (s '(lambda (x) (s x (list x)))
     (list '(lambda (x) (s x (list x))))))
```

L' "amibe" est le cas le plus simple d'autotransformation puisque la transformation en question est la transformation identique. Voilà un exemple d'autotransformation non-identique. Il s'agit d'un programme qui se "conse" à son argument :

```
(def 'new-cons '(lambda (x y) (cons y x)))

(INI (VOILA (NEW-CONS (LAMBDA (X Y) (CONS Y X)))))

(new-cons '(a b c) 'd)

(INI (D A B C))

(k new-cons)

(INI (LAMBDA (Y)
     (CONS Y
          (S '(LAMBDA (X Y) (CONS Y (S X (LIST X))))
            (LIST '(LAMBDA (X Y) (CONS Y (S X (LIST X)))))))))

((k new-cons) 'azerty)

(INI (AZERTY LAMBDA (Y)
    (CONS Y
         (S '(LAMBDA (X Y) (CONS Y (S X (LIST X))))
           (LIST '(LAMBDA (X Y) (CONS Y (S X (LIST X)))))))))
```

Remarque l'amibe de nos étang, ou son système génétique, peut être considéré comme un mot écrit dans un langage moléculaire[3]. Le message est plus sophistiqué que le simple "reproduis-moi" qui résulte de l'application de k sur le code de la fonction identique. La sémantique (au sens opérationnelle vague) d'une amibe plus sophistiquée peut être :

- maintiens-moi (et donc construis les enzymes nécessaires à mon équilibre physique et chimique) ;

-protège-moi (et donc construis et maintiens une ou plusieurs membranes autour de moi, des systèmes de défenses contre les agressions, etc..) ;

- reproduis-moi, (et donc, au moment opportun, construis les enzymes capable de me reproduire, etc...).

L'éxécution est opérée par l'univers dans laquelle une amibe, concrète et singulière, est plongée. En ce sens l'amibe est un mot, e, relativement à un univers (en l'occurrence physiquo-chimique). Avec MDI, la sophistication peut être intensionnellement capturée, de façon un peu elliptique, par les solutions des équations de récursion suivantes :

$\phi_e(x)$ = dovetelle (en fonction de x, maintiens(e) & protège(e) & reproduis (e))

---

[3] Voir Myhill 1964 une biologie abstraite fondée sur la récursion, voir aussi Hofstadter 1979 pour des comparaisons entre la génétique et la récursion. Voir Smith 1971, 1992, pour des applications de 2-REC afin de construire des automates cellulaires autoreproducteurs. Le premier automate de ce genre fut trouvé par von Neumann 1966.

Comparaison avec les paradoxes

On se rappelle du barbier Z qui ne rase l'individu X ssi X ne se rase pas lui-même.

$$ZX <-> \neg(XX)$$

alors ZZ <-> ¬(ZZ). Cela revient à démontrer par l'absurde l'inexistence d'un tel barbier. On peut reconnaître la preuve du théorème de Kleene en remplaçant les arguments par leurs descriptions.

J'illustre avec l'amibe. Pour construire un programme autoreproducteur il suffit de construire un duplicateur D. Celui-ci, appliqué à une description d'un programme (ou plus généralement d'une expression) X, donne une description (desc) de X appliqué à une description de X :

$$D(desc(X)) = desc(X\ (desc(X)) \qquad \text{première diagonalisation}$$

alors D(desc(D)), deuxième diagonalisation, donne une description de D(desc(D)). On obtient une forme autoreproductible. Si r est le code (index naturel) de P, ou avec abus : si $P = \phi_r$, la diagonale du paramétrisateur s fait l'office du duplicateur, si bien qu'appliquée sur elle-même, s(r,r), elle se *donne* elle-même.

Ce petit raisonnement informel peut aussi être traduit en LISP de façon directe en ne cherchant qu'une **forme** reproductible :

Il faut trouver (la description d') un programme transformant

```
(lambda (x) (... x ...))
```

en

```
((lambda (x) (... x ...))
      (quote
  (lambda (x) (... x ...))
     ))
```

Une telle manipulation de symboles est effectuée par

```
(def 'bootstrap          '(lambda (x)
                              (list x
                              (list (quote quote) x)))          )
```

la forme reproductible est, et est donnée par (bootstrap bootstrap)

```
((lambda (x) (list x
          (list (quote quote) x)
        ))
         (quote
 (lambda (x) (list x
          (list (quote quote) x)        )) )
```

ritournelle-devinette

La devinette suivante résume et épingle l'usage de la double diagonalisation :

si DA donne AA, si DB donne BB, si DC donne CC, que donne DD ?

2.2.4 La machine *miroir*

La transformation *t* détermine la machine *e* capable d'appliquer *t* sur elle-même : *e = (k t)*. 2-REC permet cependant de construire aisément une machine *e* capable d'appliquer sur elle-même toute transformation, qui lui est, par exemple, présentée comme entrée.

$$\exists e \quad \forall t \quad \forall y \qquad \phi_e(t,y) = \phi_t(e,y)$$

Il suffit de construire une machine appliquant l'applicateur de sa donnée sur elle-même, c-à-d l'applicateur dont on a permuté les arguments. L'application de *k* sur un tel applicateur donne la machine recherchée :

(k app)(f) = f(k app).

La conversation suivante, en Φ-LISP, illustre le point :

```
? (ini)    ; appel de Φ-LISP

 i          ; je rappelle que i

 (INI (LAMBDA (X) X))       ;est la fonction identité

 (def 'app '(lambda (x y) (y x)))       ; définition de l'applicateur

 (INI (VOILA (APP (LAMBDA (X Y) (Y X)))))

 (k app)      ; donne la machine miroir :

 (INI (LAMBDA (Y)
        (Y
          (S '(LAMBDA (X Y) (Y (S X (LIST X))))
             (LIST '(LAMBDA (X Y) (Y (S X (LIST X)))))))))

 ; appliquée à l'identité, elle se donne elle-même :

 ((k app) i)
```

```
(INI (LAMBDA (Y)
       (Y
         (S '(LAMBDA (X Y) (Y (S X (LIST X))))
            (LIST '(LAMBDA (X Y) (Y (S X (LIST X))))))))

; appliquée à n'importe quoi, elle donne n'importe quoi
; appliquée à elle-même :

((k app) '(lambda (x) (cons 'hello x)))

(INI (HELLO LAMBDA (Y)
       (Y
         (S '(LAMBDA (X Y) (Y (S X (LIST X))))
            (LIST '(LAMBDA (X Y) (Y (S X (LIST X))))))))

(lenght '(a b))          ; autre exemple

(INI (ERREUR LENGHT C-EST-QUOI-?))

(fprim length)           ; recherche de la primitive length du common-LISP sous-jaçant

(length '(a b))

(INI 2)

((k app) length)         ; la machine miroir donne sa propre longueur

(INI 3)
```

2.2.5 La *planaire* et le problème de Driesch

Case (1974) a généralisé le théorème de Kleene :

1°)Théorème de Case

Pour toute machine *t*, il existe une machine *e*, telle que

$$\phi_{\phi_{\cdot\cdot\cdot\phi_e(x_1)\cdot\cdot\cdot}(x_n)}(a) = \phi_t(e,x_1,...,x_n,a)$$

Intuitivement cela permet de construire une collection autoréférentielle de machines autoréférentielles, ou une collection de collections de telles machines, ou encore une collection de collections de collections de telles machines , etc. On étend la définition de capacité personnelle pour de telle collection de façon immédiate.

Helmholtz aurait sans doute bien aimé ce théorème puisqu'il coupe à la racine l'argument de Driesch en faveur du vitalisme.

Définition Une telle collection autoréférentielle sera appelée un tissu autoréférentiel.

Preuve du théorème de Case : (par induction)

Quand n=1, il s'agit du théorème de Kleene. En utilisant le théorème de paramétrisation on trouve un g (donné explicitement par l'application du métaprogramme s) tel que

$$\phi_{\phi_g(x,x_1\ldots x_n)}(a) = \phi_t(x,x_1,\ldots,x_n,a) \quad (*)$$

en prenant $g = \phi_s(t,x,x_1,\ldots,x_n)$.

Supposons que le théorème soit vrai pour n-1, on peut l'appliquer à $\phi_g(x,x_1,\ldots,x_n)$ avec x_n jouant le rôle de u. Ainsi existe-t-il un *e* tel que :

$$\phi_{\phi_{\ldots_{\phi_e(x_1)}\ldots}(x_{n-1})}(x_n) = t(e,x_1,\ldots,x_n)$$

Substituant ce résultat dans (*) donne ce qu'on voulait démontrer. QED.

2°) la planaire élémentaire

Au moyen de cette généralisation, il est à présent possible de construire une *planaire* (voir Buchsbaum 1938, Buchsbaum & Al. 1987). J'illustre la plus simple des *planaires*. Il s'agit d'une collection de deux cellules (C1,C2). Chaque cellule est capable de reconstituer la planaire entière lorsqu'on lui présente un argument particulier fixé : FOO. Donc :

$$C1(FOO) = (C1,C2)$$
$$C2(FOO) = (C1,C2)$$

J'attribue différentes fonctions, au sens extensionnel[4], à chaque *cellules :*

$$C1 = \lambda x.x+1$$
$$C2 = \lambda x.x+2$$

Construction : En utilisant le théorème de Case avec n = 2 :

$$\forall t \exists e \quad \phi_{\phi_e(y)}(z) = \phi_t(e,y,z) \quad (**)$$

ϕ_e sera un générateur de C1 et C2: $\phi_e(1) = C1$, $\phi_e(2) = C2$. On veut donc:

[4] La différence entre intensionnelle et extensionnelle est épinglée sur cet exemple.

$$\phi_{\phi_e(1)}(z) = \text{si } z=FOO \text{ alors sortir } (\phi_e(1), \phi_e(2)) \text{ sinon } z+1$$

$$\phi_{\phi_e(2)}(z) = \text{si } z=FOO \text{ alors sortir } (\phi_e(1), \phi_e(2)) \text{ sinon } z+2$$

e sera exhibé en appliquant (**) sur la matrice de récursion suivante :

$t(x,1,FOO) = (\phi_x(1), \phi_x(2))$
$t(x,2,FOO) = (\phi_x(1), \phi_x(2))$
$t(x,1,z) = z+1$, if $z \neq FOO$
$t(x,2,z) = z+2$, if $z \neq FOO$
$t(x,y,z) =$ 'error if $y > 2$.

c'est-à-dire en appliquant k et le théorème de paramétrisation à la façon de la démonstration du théorème de récursion de Case.

Je le montre dans Φ-LISP, j'utilise "m" plutôt que "t" pour éviter la confusion avec la constante booléenne prédéfinie t (le vrai).

```
(def 'm  '(lambda (x y z)
      (cond
        ((and (equal y 1) (equal z 'FOO))
         (list (x 1) '***** (x 2) ))
        ((and (equal y 2) (equal z 'FOO))
         (list (x 1) '+++++ (x 2) ))
        ((equal y 1) (+ z 1))
        ((equal y 2) (+ z 2))
        (t '(erreur:il n'y a que 2 cellules))
       ))
   )
```

Ensuite, on définit g :

```
(def 'g '(lambda (x y) (s m (list x y))) )
```

Et l'application de k sur g donne p le générateur des cellules de la planaire, par exemple (p 1) donne la première cellule.

```
(def 'p (k g)).
```

Chaque cellule génère la planaire entière lorsqu'elle est appliquée à FOO, ainsi :

((p 1) 'FOO)[5]

donne la planaire entière (voir l'illustration en Φ-LISP plus bas).
et chaque cellule fonctionne correctement :

((p 1) 3)

4

((p 2) 3)

5

Cette planaire a une capacité personnelle par rapport à l'(auto)régénération relativement à LISP.

Illustration en Φ-LISP

```
? (ini)

(def 'F-planaire  '(lambda (x y z)
      (cond
        ((and (equal y 1) (equal z 'FOO))
           (list (x 1) '********** (x 2) ))
        ((and (equal y 2) (equal z 'FOO))
          (list (x 1) '++++++++++++++ (x 2) ))
        ((equal y 1) z)
        ((equal y 2) (+ z 1))
        (t 'ERREUR)
       ))
   )

(INI (VOILA (F-PLANAIRE (LAMBDA (X Y Z) (COND ((AND (EQUAL Y 1) (EQUAL Z 'FOO)) (LI
ST (X 1) '********** (X 2))) ((AND (EQUAL Y 2) (EQUAL Z 'FOO)) (LIST (X 1) '+++
++++++++++ (X 2))) ((EQUAL Y 1) Z) ((EQUAL Y 2) (+ Z 1)) (T 'ERREUR))))))

(def 'G-planaire '(lambda (x y) (s F-planaire (list x y))) )

(INI (VOILA (G-PLANAIRE (LAMBDA (X Y) (S F-PLANAIRE (LIST X Y))))))

(def 'E-planaire (k G-planaire))

(INI (VOILA (E-PLANAIRE (LAMBDA (Y) (S F-PLANAIRE (LIST (S '(LAMBDA (X Y) (S F-P
LANAIRE (LIST (S X (LIST X)) Y))) (LIST '(LAMBDA (X Y) (S F-PLANAIRE (LIST (S X
(LIST X)) Y))))) Y))))))

(pp ((e-planaire 1) 'FOO))

((LAMBDA (Z)
  (COND ((AND (EQUAL '1 1) (EQUAL Z 'FOO))
        (LIST ('(LAMBDA (Y)
                  (S F-PLANAIRE
                    (LIST (S '(LAMBDA (X Y)
                                (S F-PLANAIRE (LIST (S X (LIST X)) Y)))
                             (LIST '(LAMBDA (X Y)
                                      (S
                                        F-PLANAIRE
                                        (LIST (S X (LIST X)) Y)))))
                             Y)))
              1)
```

[5] Il faut regarder ce code comme un biologiste regarde une molécule de DNA.(Marchal 1992).

```
          '**************
        ('(LAMBDA (Y)
           (S F-PLANAIRE
              (LIST (S '(LAMBDA (X Y)
                          (S F-PLANAIRE (LIST (S X (LIST X)) Y)))
                       (LIST '(LAMBDA (X Y)
                                (S
                                  F-PLANAIRE
                                   (LIST (S X (LIST X)) Y)))))
                       Y)))
           2)))
      ((AND (EQUAL '1 2) (EQUAL Z 'FOO))
       (LIST ('(LAMBDA (Y)
                 (S F-PLANAIRE
                    (LIST (S '(LAMBDA (X Y)
                                (S F-PLANAIRE (LIST (S X (LIST X)) Y)))
                             (LIST '(LAMBDA (X Y)
                                      (S
                                        F-PLANAIRE
                                         (LIST (S X (LIST X)) Y)))))
                             Y)))
           1)
           '++++++++++++++
           ('(LAMBDA (Y)
              (S F-PLANAIRE
                 (LIST (S '(LAMBDA (X Y)
                             (S F-PLANAIRE (LIST (S X (LIST X)) Y)))
                          (LIST '(LAMBDA (X Y)
                                   (S
                                     F-PLANAIRE
                                      (LIST (S X (LIST X)) Y)))))
                          Y)))
           2)))
      ((EQUAL '1 1) Z)
      ((EQUAL '1 2) (+ Z 1))
      (T 'ERREUR)))
 ***************
(LAMBDA (Z)
  (COND ((AND (EQUAL '2 1) (EQUAL Z 'FOO))
         (LIST ('(LAMBDA (Y)
                   (S F-PLANAIRE
                      (LIST (S '(LAMBDA (X Y)
                                  (S F-PLANAIRE (LIST (S X (LIST X)) Y)))
                               (LIST '(LAMBDA (X Y)
                                        (S
                                          F-PLANAIRE
                                           (LIST (S X (LIST X)) Y)))))
                               Y)))
           1)
            '**************
           ('(LAMBDA (Y)
              (S F-PLANAIRE
                 (LIST (S '(LAMBDA (X Y)
                             (S F-PLANAIRE (LIST (S X (LIST X)) Y)))
                          (LIST '(LAMBDA (X Y)
                                   (S
                                     F-PLANAIRE
                                      (LIST (S X (LIST X)) Y)))))
                          Y)))
           2)))
      ((AND (EQUAL '2 2) (EQUAL Z 'FOO))
       (LIST ('(LAMBDA (Y)
                 (S F-PLANAIRE
                    (LIST (S '(LAMBDA (X Y)
                                (S F-PLANAIRE (LIST (S X (LIST X)) Y)))
                             (LIST '(LAMBDA (X Y)
                                      (S
                                        F-PLANAIRE
                                         (LIST (S X (LIST X)) Y)))))
```

```
                Y)))
            1)
            '+++++++++++++++
            ('(LAMBDA (Y)
               (S F-PLANAIRE
                  (LIST (S '(LAMBDA (X Y)
                              (S F-PLANAIRE (LIST (S X (LIST X)) Y)))
                           (LIST '(LAMBDA (X Y)
                                    (S
                                     F-PLANAIRE
                                      (LIST (S X (LIST X)) Y)))))
                           Y)))
               2)))
         ((EQUAL '2 1) Z)
         ((EQUAL '2 2) (+ Z 1))
         (T 'ERREUR))))
(INI NIL)
```

On voit que chaque cellule de ce réseau autoréférentiel est un ***oeuf potentiel*** en ce sens que chaque cellule est capable de régénérer, sur quelque donnée fixée (comme FOO) le réseau dans son entièreté, comme l'oeuf, la planaire et la grenouille rencontrée dans la première partie ; l'identité est distribuée dans l'entièreté du tissu. L'existence et la construction exhibée par l'application de k, réfutent les preuves de l'impossibilité de l'auto-reproduction qui invoque l'apparition d'une régression infinie[6].

Remarquons également que chaque cellule possède l'information que possèdent toutes les autres cellules. Une immense redondance apparaît au sein de l'organisme (vu comme une collection autoréférentielle de cellules autoréférentielles). mais tel est en fait le cas chez les organismes biologiques où chaque cellule d'un organisme possède le code de tout l'organisme[7], ce qui a été montré par les généticiens (voir Watson 1965[8]).

Case avait déjà observé la compatibilité entre la reproduction biologique, le second théorème de recursion et la solution de Von Neumann au problème de Descartes (voir von Neumann 1966, voir aussi Smith 1971, 1992, Case 1971, 1974).

3°) construction de cellules moins redondantes

On aura constater une forte redondance dans les codes obtenus par l'application de k.

Une opération élémentaire de *lambda calcul* (ou d'informatique) permet de construire des cellules moins redondantes

Si on diagonalise l'expression suivante par rapport à x

---

6 Rogers avait déjà remarqué ce point (Rogers 1967, page 189), mais en fait Post 1921 aussi, voir la footnote 112 à la page 424 du Davies 1965. Il ne faut pas remonter à Descartes et à Driesch pour constater de telles invocations, voir par exemple Cossa 1955, ou Raymond Ruyer 1966.

7 En biologie tout principe admet une exception en l'occurrence ici, par exemple, les globules rouges qui sont dépourvus de noyaux, et donc de matériel génétique.

8 La première édition de 1965, écrite par Watson seul, est bien suffisante pour notre propos. Elle est avantageuse en ce qui concerne son poids!

$$\lambda xyzf(g(x), h(x),...^{<n \text{ occurrences de } x>} ..y, z),$$

alors les n occurrences de x sont multipliées (physiquement) par deux. En diagonalisant la diagonale, k substitue chaque occurrence de x par l'expression de départ. Si le nombre d'occurrences de x est n au départ on obtient $(2n)^2$ occurrences à l'arrivée.

En informatique l'idée pour supprimer cette redondance est d'utiliser un LET

$$\text{LET } r = x \text{ ; } f(g(r),h(r), ..y,z)$$

Avec λ, on peut simuler ce LET :

$$\lambda x((\lambda ryz\ f(g(r), h(r),...^{<n \text{ occurrences de } x>} ..)x)$$

le nombre de x est d'office 1 au départ, la redondance est d'office d'ordre $4 = (2*1)^2$,

appliquée à la planaire, cela donne

```
(def 'F-planaire-symbole  '(lambda (x y z)
   ((lambda (r y z)
      (cond
        ((and (equal y 'A) (equal z nil))
          (list  (r 'A) '********** (r 'B) ))
        ((and (equal y 'B) (equal z nil))
          (list (r 'A) '++++++++++++++ (r 'B) ))
        ((equal y 'A) (car z))
        ((equal y 'B) (car (cdr z)))
        (t 'ERREUR)))
       x y z    ))

(def 'G-planaire-symbole '(lambda (x y) (s F-planaire-symbole (list x y))) )

(def 'E-planaire-symbole (k G-planaire-symbole))


(pp ((e-planaire-symbole 'a) nil))

((lambda (z)
  ((lambda (r 'a z)
     (cond ((and (equal 'a 'a) (equal z nil))
          (list (r 'a) '********** (r 'b)))
         ((and (equal 'a 'b) (equal z nil))
          (list (r 'a) '++++++++++++++ (r 'b)))
         ((equal 'a 'a) (car z))
         ((equal 'a 'b) (car (cdr z)))
         (t 'erreur)))
   '(lambda (y)
      (s f-planaire-symbole
       (list (s '(lambda (x y) (s f-planaire-symbole (list (s x (list x)) y)))
            (list '(lambda (x y)
                     (s f-planaire-symbole (list (s x (list x)) y)))))
          y)))
```

```
  'a z))
**************
(lambda (z)
  ((lambda (r 'b z)
     (cond ((and (equal 'b 'a) (equal z nil))
            (list (r 'a) '************** (r 'b)))
           ((and (equal 'b 'b) (equal z nil))
            (list (r 'a) '+++++++++++++++ (r 'b)))
           ((equal 'b 'a) (car z))
           ((equal 'b 'b) (car (cdr z)))
           (t 'erreur)))
   '(lambda (y)
      (s f-planaire-symbole
       (list (s '(lambda (x y) (s f-planaire-symbole (list (s x (list x)) y)))
             (list '(lambda (x y)
                      (s f-planaire-symbole (list (s x (list x)) y)))))
             y)))
   'b z)))

(pp ((e-planaire-symbole 'a) '(aa bb cc)))
AA
(INI NIL)
(pp ((e-planaire-symbole 'b) '(aa bb cc)))
BB
(INI NIL)
(pp ((e-planaire-symbole 'c) '(aa bb cc)))
ERREUR
(INI NIL)
(fini)
(TERMINE INI)
NIL
```

Ceci illustre une transformation intensionnelle qui respecte l'extension *et* l'intension du programme initial relativement à un environnement.

Ici aussi une planaire plus sophistiquée peut être présentée, en distribuant non seulement une capacité régénérative, mais des intensions plus générales, jouant le rôle d'intentions (avec t) instinctives, si on veut :

$$\phi\phi_e(x)(y) = (\text{dovetelle en fonction de y, maintiens}(\phi_e(x)) \;\&\; \text{protège}(\phi_e(x)) \;\&\; \text{et reproduis}(\phi_e(x))$$

4°) Planaire infinie

On construit une suite infinie de machines m_i, telle que m_i(next) donne m_i+1, m_i(before) donne m_i-1 (sauf si $i = 0$), et m_i (<autre chose que next>) donne m_i. Comme toutes les m_i peuvent être identiques, on utilisera le *padding*[9] pour les différencier explicitement, ou on essayera plus

[9]Pour tout programme il existe un programme (syntaxiquement) plus grand qui calcule la même fonction $\forall i \exists j\ j > i \;\&\; \phi_i = \phi_j$, (la façon la plus simple de s'en convaincre consiste pour construire un tel j d'ajouter dans i des instructions inutiles).

simplement de leur faire faire quelque chose de différent chacune, par exemple : imprimer son propre numéro *i* sur l'argument 'number :

```
(def 'm
  '(lambda (x i z)
    (cond ((and (equal i 0) (equal z 'before)) (PP 'NOBEFORE))
          ((equal z 'next) (x (+ i 1)))
          ((equal z 'before) (x (- i 1)))
          ((equal z 'number) i)
          (t (x i)))))

(def 'g '(lambda (x y) (s m (list x y))))

(def 'p (k g))
```

2.2.6 Applications typiques

1°) Simple récursion et récursion anonyme

Jusqu'ici les machines digitales (les "programmes") produisent des déformations statiques d'elles-mêmes. Rien ne leur interdit cependant d'invoquer la machine universelle capable de les exécuter dynamiquement :

$$\phi_e(x) = \phi_u(e,x)$$

Cela permet de définir des dynamiques récursives, y compris les partielles de la famille du fameux *10 GOTO 10* du BASIC :

```
(def 'infini-F '(lambda (x) (u x))

(def 'infini '(k infini-F))
```

Ceci illustre aussi l'évident fait que la machine universelle est elle-même partielle.

Autre exemple : une factorielle *anonyme*

```
(setq fa   '(
  (def 'F-fact '(lambda (y x)
        (cond ((equal x 0) 1)
              (t (* x (y (- x 1) ) ))  ) ))   ; ou (* x (u y (- x 1)))
    ))

(def 'F-fact '(lambda (y x)
        (cond ((equal x 0) 1)
              (t (* x (y (- x 1) ) ))  ) ))

(INI (VOILA (F-FACT (LAMBDA (Y X) (COND ((EQUAL X 0) 1) (T (* X (Y (- X 1)))))))
))

(pp (k f-fact))
```

```
(LAMBDA (X)
 (COND ((EQUAL X 0) 1)
    (T
     (* X
        ((S '(LAMBDA (Y X)
             (COND ((EQUAL X 0) 1) (T (* X ((S Y (LIST Y)) (- X 1))))))
           (LIST '(LAMBDA (Y X)
                (COND ((EQUAL X 0) 1)
                   (T (* X ((S Y (LIST Y)) (- X 1))))))))
         (- X 1))))))
(INI NIL)

((k f-fact) 10)
(INI 3628800)
```

Pour la récursion multiple on peut utiliser la généralisation de Case (voir pair-impair, voir une paire eval apply obtenue par une procédure plus uniforme d'*anonymisation* en annexe 1).

2°) le théorème du point fixe

Tous les programmes peuvent jouer le rôle de métaprogramme. Un métaprogramme *i* est une opération de l'ensemble de tous les programmes sur lui-même. Le théorème du point fixe énonce que tous les métaprogrammes laissent toujours le *comportement* (l'extension) d'un programme invariant :

$$\forall i \, \exists e \, \forall x \;\; \phi_{\phi_i(e)}(x) = \phi_e(x)$$

preuve

Quel que soit i, il suffit d'appliquer 2-REC sur la fonction (éventuellement partielle) $g(x,y) = \phi_{\phi_i(x)}(y) = \phi_t(x,y)$. (En fait la transformation $t = \ulcorner \lambda x \lambda y \, \phi_{\phi_i(x)}(y) \urcorner$). En effet 2-REC affirme l'existence d'un e tel que $\phi_e(y) = \phi_t(e,y) = \phi_{\phi_i(e)}(y)$, ce qu'on voulait.

Sur les énumérations des fonctions partielles calculables, le théorème du point fixe est équivalent au théorème de récursion de Kleene : $\forall t \exists e \; \phi_e(y) = \phi_t(e,y)$. En effet, il suffit d'appliquer le métaprogramme de paramétrisation s sur $\phi_t(x,y) = \phi_{\phi_s(t,x)}(y)$. La fonction de x, $\phi_s(t,x) = \phi_i(x)$, admet un point fixe e : $\phi_e(x) = \phi_{\phi_i(e)}(x) = \phi_{\phi_s(t,e)}(x) = \phi_t(e,y)$.

C'est la raison pour laquelle Rogers présente le théorème de récursion sous la forme du théorème du point fixe.

Cependant on peut montrer que les énumérations des écoles du dedans (sub-universelles) satisfont le second théorème de récursion (2-REC) et ne satisfont pas le théorème du point fixe. En effet pour calculer $\lambda x \lambda y \phi_{\phi_i(x)}(y)$,

on doit calculer une fonction universelle $\phi_u(i,x)$, et seulement si ce calcul converge, vers k par exemple, peut-on calculer $\phi_k(y)$, mais les écoles du dedans n'ont pas de machines universelles. De telles machines universelles peuvent exister, mais ne peuvent pas être décrites du dedans sous peine d'être fermée pour la diagonalisation et d'admettre alors une bête partielle. 2-REC, quant à lui ne repose pas sur l'existence d'une fonction universelle, seulement de l'universalité paresseuse[10].

3°) une conséquence fondamentale de 2-REC (théorème de Rice)

définition La fonction partielle ϕ_i est *extensionnellement* égale à la fonction ϕ_j, on écrit $\phi_i = \phi_j$, si elles sont définies sur le même domaine et y ont les mêmes valeurs :

$$\forall x \{(\phi_i(x)\Downarrow \& \phi_j(x)\Downarrow \text{ -> } \phi_i(x) = \phi_j(x)) \text{ v } \phi_i(x)\Uparrow \& \phi_j(x)\Uparrow\}$$

Les programmes *i* et *j* ont les mêmes entrées/sorties. A quelques abstractions près (le temps d'exécution notamment) on peut dire que les machines *i* et *j* ont le même comportement extensionnel. Dit encore autrement : les nombres *i* et *j* définissent (relativement à un univers *u*) la même transformation partielle de ω dans ω.

Définition :
i et *j* sont extensionnellement équivalents, j' écris $i \equiv j$, si $\phi_i = \phi_j$.

Théorème "≡" n'est pas décidable (Rice).
preuve (sketch) si ≡ était décidable, il serait aiser de construire une permutation sans *points fixes* de l'ensemble des machines.

On a immédiatement :
Théorème ∀i l'ensemble $\{j \mid \phi_i = \phi_j\}$ n'est pas récursif.

Le contraire aurait été remarquable car on aurait pu mécaniquement résoudre Fermat en testant l'appartenance ou la non-appartenance du code correspondant à

f(x) = chercher une solution pour Fermat, et si on en trouve une, alors sortir factoriel (x), sinon continuer à chercher.

---

[10] C'est-à-dire d'une machine paramétrisatrice. On peut voir une machine paramétrisatrice *s* comme une sorte de machine universelle paresseuse, elle ne procède qu'à des substitutions (ou même des concaténations pour certaines énumérations acceptables spéciales). De cette façon, elle se contente de décrire tous les calculs possibles sans rien vraiment éxécuter (d'où son nom). Voir aussi Royer1987, Kurtz, Mahaney, Royer 1990.

dans l'ensemble $\{i \mid \phi_i = \text{factoriel}\}$

De même, le prédicat $A(x,y,z) <-> \phi_x(y) = z$, n'est pas récursif.

4°) ordinaux constructifs.

Pour voyager dans un ensemble productif, et de façon plus particulière dans le complémentaire d'un ensemble créatif, il est nécessaire de disposer d'une bonne échelle transfinie. Pour ça faut-il disposer d'un bon système de notation pour les ordinaux. Existe-t-il un système de notation maximale, c-à-d capable de nommer tous les ordinaux nommables par les machines ? A première vue la réponse est non, puisqu'un tel système désignerait un ordinal nommable par la machine. C'est encore 2-REC, qui permet de définir un tel système maximal sans mettre en branle un principe de Watts-Valadier *local*. Plutôt que d'exposer la théorie qu'on trouvera au chapitre 11 de Rogers 1967, j'illustre brièvement le principe.

k-ck est un prédicat calculable qui distingue 0, les notations des ordinaux successeurs, les notations pour les ordinaux limites :

(k-ck 0) = 1

(k-ck x) = 2 si x est une notation pour un ordinal successeur,

(k-ck x) = 3 si x est une notation pour un ordinal limite.

p-ck est une fonction prédecesseur, c-à-d que si x est une notation pour un ordinal successeur, p-ck(x) donne une notation pour son (unique) prédécesseur, q-ck est telle que si x est une notation pour un ordinal limite L alors la suite ((q-ck x) 1), ((q-ck x) 2), ... sont des notations pour une suite fondamentale convergeant vers L.

En utilisant l'opérateur de Kleene, on montre que c'est encore la fermeture du dehors pour la diagonalisation qui empêche l'utilisation de principe de Watts-Valadier *local* pour réfuter l'existence d'un plus petit ordinal non constructif. J'illustre en LISP un tel système.

(1) est la notation pour 0,

(2 x) est une notation pour a + 1, si x est une notation pour a,

(3 e) est une notation pour l'ordinal limite λ si

(lambda (x) (e x))

défini sur N une suite fondamentale pour λ. C'est-à-dire une suite d'ordinaux dont la limite est λ. En fait (3 e) peut servir de système de notation pour les ordinaux inférieur à λ.

avec

```
(def 'k-ck '(lambda (o) (p1 o)))
(def 'p-ck '(lambda (o) (p2 o)))
(def 'q-ck '(lambda (o) (p2 o)))
```

ici p-ck = q-ck.

exemples en Φ-LISP :

(1) est une notation pour 0

(2 (2 (2 (1)))) est une notation pour 3

(list 3 (k '(lambda (y x) (cond ((zéro? x) '(1))(t (cons 2 (list (y (moins x 1)))))))))) donne une notation pour OMEGA :

c-à-d

```
(3 (lambda (x)
  (cond ((zéro? x) '(1))
      (t
       (cons 2
          (list ((s
                '(lambda (y x)
                   (cond ((zéro? x) '(1))
                     (t
                        (cons 2 (list ((s y (list y)) (moins x 1)))))))
                (list '(lambda (y x)
                       (cond ((zéro? x) '(1))
                         (t
                           (cons 2
                              (list
                                ((s y (list y))
                                 (moins x 1)))))))))
                                 (moins x 1)))))))
```

remarque : si on avait utilisé les nombres naturels comme notation (3 i) aurait été une notation pour oméga.

(2 (3 omega)) est une notation pour oméga + 1

avec

```
(def 'o+o (k '(lambda (y x)
            (cond ((zéro? x) (list 3 omega))
                (t (cons 2 (list (y (moins x 1)))))
              ))))
```

(2 (2 (2 (3 o+o)))) est une notation pour ω+ω+3

c-à-d, avec les détails :

```
(pp (o+o 3))
```

```
(2
 (2
  (2
   (3
    (lambda (x)
      (cond ((zéro? x) '(1))
            (t
             (cons 2
                 (list ((s
                       '(lambda (y x)
                          (cond ((zéro? x) '(1))
                                (t
                                 (cons 2
                                     (list
                                         ((s y (list y)) (moins x 1)))))))
                       (list '(lambda (y x)
                                (cond ((zéro? x) '(1))
                                      (t
                                       (cons
                                       2
                                        (list
                                           ((s y (list y)) (moins x 1)))))))))
                       (moins x 1)))))))))))
```

Définition ω_1^{CK} est le plus petit ordinal non constructif. La thèse de Church confère un statut absolu (intuitif) à cet ordinal. Les plus petits ordinaux non explicitement définissables en FORTRAN, algol, LISP, c++, mais aussi quelques patterns du *jeu de la vie*, certains problèmes de n corps, les équations diophantiennes universelles, etc., sont tous identiques et égaux à ω_1^{CK}.

En outre, différentes définitions des ordinaux constructifs se sont révélées équivalentes ce qui confirme à la fois l'intérêt de la thèse de Church et l'adéquation de la définition des ordinaux constructifs par Church et Kleene. ω_1^{CK} est le chemin transfini sur laquelle les machines sont prouvablement (et formellement) capables de se déplacer (à l'exception de ω_1^{CK} lui-même).

A ce sujet, existe-t-il une machine capable d'escalader l'échelle de tous les ordinaux constructifs. Cela n'est pas impossible. Ce qui est impossible, c'est de prouver au sujet d'une machine particulière qu'elle est capable d'escalader cette échelle. Si une ω_1^{CK}-machine existe, elle ne peut pas exister constructivement. Son existence ne peut pas être prouvée de façon intuitioniste. elle appartient résolument à l'espace du dehors.

A nouveau une promesse potentielle, une possibilité, est offerte au prix d'une impossibilité algorithmique. Une machine peut grimper l'entièreté de ω_1^{CK}, mais pas de façon prouvable.

Une machine universelle peut être vue comme un nom *implicite* pour ω_1^{CK}. Un tel système peut donner des noms à des ordinaux α, β sans qu'il soit mécaniquement possible de décider si $\alpha > \beta$, ou $\beta < \alpha$.

Remarques

1) On peut lier les ordinaux constructifs avec les écoles du dedans définies par l'ensemble des fonctions prouvablement calculables dans une théorie donnée, en mesurant la puissance de cette théorie au moyen de son analyse ordinal.

2) Si SN est un système de notation pour les ordinaux constructifs récursivement relatés ("$\alpha > \beta$" est récursif), alors il existe un ordinal constructif non nommé par SN, (et il existe un système SN' qui le nomme, on peut le trouver uniformément à partir de SN).

5°) Le castor occupé[11] *BB (buzy beaver)*

Fixons U un langage de programmation, ou une machine universelle. J'écris U(x) pour $\phi_u(x)$, x étant compris comme parcourant les descriptions des programmes à 0 entrée.

Définition : *BB(n)* est le plus grand nombre qu'on peut écrire avec un programme de longueur inférieure ou égale à n :

$$BB(n) = \max \{m \mid \exists p \text{ longueur}(p) \leq n \ \& \ U(p) = m\}$$

Théorème : aucune machine n'est capable de calculer *BB*.

preuve : en effet supposons que *b* calcule *BB* (qui est total), alors, avec 2-REC, on peut construire le programme *e* tel que $\phi_e() = 2^{\phi_b(longueur(e))}$

BB est un exemple d'une fonction totale non calculable. Ou d'un schéma de fonction non calculable dépendant du système universel correspondant à la numérotation $\{\phi_x\}$.

Théorème Le problème de l'arrêt est réductible en *BB*. (avec un oracle pour *BB* je sais décider de l'arrêt)

Preuve . En effet, supposons qu'on veuille savoir si p s'arrête, on construit p', tel que U(p') = le nombre de step de p. On a p s'arrête ssi p' s'arrête. On lance U(p), si U(p) s'arrête en un nombre de step plus petit ou égale à BB(p'), on sait évidemment que p s'arrête. Si le nombre de step de U(p) dépasse BB(p'), on sait que p ne s'arrêtera plus. En effet s'il s'arrête, alors U(p') s'arrêterait sur un nombre plus grand que BB(p'). QED.

BB donne ainsi pour toute machine m un nombre n tel que si m ne s'est pas arrêtée avant n étape, elle ne s'arrêtera plus. Cela justifie qu'en observant sufisamment longtemps les machines on se rapproche du réel de Post-Turing.

---

[11] Je ne donne pas la définition standard (en terme d'états de machine de Turing).

Inversément, si on dispose d'un oracle pour ρ on sait calculer BB, c-à-d les solutions au problème de l'arrêt pour les fonctions à un argument on sait facilement calculer BB vu que le nombre de programmes p de longueur ≤ n est fini, avec ρ on rejette ceux qui ne s'arrêtent pas, on lance tous les autres, ils vont tous s'arrêter forcément, à ce moment-là on prend la plus grande sortie (output) donné par un de ces programmes.

Conclusion : un oracle pour *BB* est aussi fort qu'un oracle pour ρ et réciproquement.

Corollaire : une théorie de longueur *n* ne sait pas prouver l'arrêt de programme dont le nombre de step dépasse BB(n) (voir aussi Chaitin 1987).

6°) l'insolubilité du problème de l'arrêt d'une machine

a) avec BB

b) directement avec 2-REC.

c) par diagonalisation élémentaire (voir Rogers 1967).

2.2.7 Machines *introspectives*

Une machine peut-elle décrire son *comportement intensionnel?* Il est défini par la description de la suite de ses propres états relativement à un univers *u* (avec ou sans oracle).

Posé autrement : un programme peut-il décrire intégralement sa propre trace ? Existe-t-il un programme capable de décrire sa trace[12] ?

1°) Solution 1

La solution directement tirée de 2-REC, c'est-à-dire la solution de l'équation

$$\phi_e() = \phi_{trace}{}^{13'}(e),$$

donnée par (k trace). L'éxécution de e, $\phi_{(k\ trace)}() = ((k\ trace))$ donne la trace de e.

Remarquons que ce n'est pas le programme que l'on trace, mais une forme, c'est-à-dire la description de l'application du programme sur un argument (ici vide).

---

[12] Voir aussi le chapitre 11 de Cutland 1980.

[13] Voir 2. 1 pour la définition de trac, et de trac-exec.

Donc e doit donner la trace de (e). Il suffit d'appliquer l'opérateur de Kleene k sur un programme qui, appliqué à x, évalue la trace de (x), c-à-d (list x) :

```
(def 'trac-moi-f '(lambda (x) (trac (list x))))
(def 'trac-moi ((k trac-moi-f)))
```

trac , qui appliqué à une forme donne la trace de son éxécution, peut être défini en Common Lisp et être rendu primitif en Φ-LISP avec l'instruction (fprim trac). Dans ce cas E-step lui-même ne s'arrête pas lors de l'appel effectif de la trace sur son argument, c'est l'étape 6 ci-dessous.

```
1
(TRAC (LIST (S '(LAMBDA (X) (TRAC (LIST (S X (LIST X)))))
          (LIST '(LAMBDA (X) (TRA C (LIST (S X (LIST X)))))))))

2
(TRAC (LIST (S (VAL (LAMBDA (X) (TRAC (LIST (S X (LIST X))))))
          (LIST '(LAMBDA (X) (TRAC (LIST (S X (LIST X)))))))))

3
(TRAC (LIST (S (VAL (LAMBDA (X) (TRAC (LIST (S X (LIST X))))))
          (LIST (VAL (LAMBDA (X) (TRAC (LIST (S X (LIST X)))))))))

4
(TRAC (LIST (S (VAL (LAMBDA (X) (TRAC (LIST (S X (LIST X))))))
          (VAL ((LAMBDA (X) (TRAC (LIST (S X (LIST X)))))))))

5
(TRAC (LIST (VAL
         (LAMBDA NIL
           (TRAC (LIST (S '(LAMBDA (X) (TRAC (LIST (S X (LIST X)))))
                     (LIST '(LAMBDA (X)
                               (TRAC (LIST (S X (LIST X))))))))))))

6
(TRAC (VAL
     ((LAMBDA NIL
       (TRAC (LIST (S '(LAMBDA (X) (TRAC (LIST (S X (LIST X)))))
                 (LIST '(LAMBDA (X) (TRAC (LIST (S X (LIST X))))))))))))

7
(TRAC (LIST (S '(LAMBDA (X) (TRAC (LIST (S X (LIST X)))))
          (LIST '(LAMBDA (X) (TRA C (LIST (S X (LIST X)))))))))
```

L'étape 7 est identique à l'étape 1, la machine ne peut plus s'arrêter.

La fonction trace n'appartenant à aucune école du dedans, le non-arrêt, donc la partialité, du programme qui observe l'entièreté de sa trace est nécessaire. Il en est de même si trac est défini explicitement dans Φ-LISP, et dans ce cas E-step est toujours bien défini. La période de la trace est alors beaucoup plus grande.(voir annexe).

Si on veut que le programme introspectif s'arrête, on va devoir se contenter d'une *exploration limitée* de la trace (solution 2), ou d'une *description* de la trace (solution 3).

2°) solution 2

On applique le métaprogramme de Kleene sur une fonction, trac-exec, qui appliquée à une forme et un nombre n, génère les n premières étapes de la trace de l'éxécution de la forme :

```
(def 'trace-moi-F '(lambda (x n) (trac-exec  (list x n) n)))
(def 'trace-moi (k trace-moi-F))
```

ce qui donne :

```
(LAMBDA (N)
      (TRAC-EXEC (LIST (S '(LAMBDA (X N)
                              (TRAC-EXEC (LIST (S X (LIST X)) N) N))
                           (LIST '(LAMBDA (X N)
                                    (TRAC-EXEC (LIST (S X (LIST X)) N) N))))
                         N)
                   N))
```

A priori, (trace-moi <n>) devrait toujours s'arrêter. En effet, appliqué à une forme dont l'éxécution est infinie, (trace-exec <n>) donne la liste des n premières étapes suivie du message que le nombre d'étapes est plus grand que N. Toutefois, si trace-exec est définie comme primitive, le nombre d'étapes N de (trace-moi <n>) ne dépendra pas de n. (trace-moi <n>) s'arrêtera pour toutes les valeurs de n inférieure à N, et divergera pour les valeurs supérieures ou égales à n. Si ce n'était pas le cas (trace-moi N) donnerait l'intégralité de sa trace. En fait, comme dans la première solution, le caractère primitif de trac-exec fera diverger E-step sur

```
(  (LAMBDA (N)
      (TRAC-EXEC (LIST (S '(LAMBDA (X N)
                              (TRAC-EXEC (LIST (S X (LIST X)) N) N))
                           (LIST '(LAMBDA (X N)
                                    (TRAC-EXEC (LIST (S X (LIST X)) N) N))))
                         N)
                   N))   <i>)
```

avec i plus grand que N. Que vaut N ? blum est une fonction qui calcule le nombre d'étapes de l'éxécution d'une forme.

```
(blum '(trace-moi 1))
(INI 7)
(blum '(trace-moi 2))
(INI 7)
(blum '(trace-moi 3))
(INI 7)
```

```
(blum '(trace-moi 4))
(INI 7)
(blum '(trace-moi 5))
(INI 7)
(blum '(trace-moi 6))
(INI 7)
(blum '(trace-moi 7))
Aborted
```

Comme avec la solution 1, si trac-exec n'est pas primitive, et donc par exemple définie directement dans Φ-LISP, le nombre d'étapes va croître en fonction de n, et (trace-moi <n>) va s'arrêter pour tout n, celui-ci étant toujours plus petit que (blum (trace-moi <n>))

```
(blum '(trace-moi 1))
(INI 34)
(blum '(trace-moi 2))
(INI 49)
(blum '(trace-moi 3))
(INI 64)
(blum '(trace-moi 4))
(INI 79)
(gc)
(INI (ERREUR GC C-EST-QUOI-?))
(fprim gc)                    voilà comment garbage-collecter en Φ-LISP
(INI (GC PPTR
       CONS

       FPRIM
       DPRIM))
(gc)
(INI NIL)
(blum '(trace-moi 5))
(INI 94)
(blum '(trace-moi 6))
(INI 109)
(blum '(trace-moi 7))
(INI 124)
(blum '(trace-moi 8))
(INI 139)
(blum '(trace-moi 9))
(INI 154)
(blum '(trace-moi 10))
(INI 169)
```

```
(blum '(trace-moi 11))
(INI 184)
```

En utilisant PPTR, un programme qui recherche les périodes dans les traces, on peut voir que la période de (trace-moi 10) vaut 19. (Trace-moi <n>) s'arrête pour n supérieure à sa période :

```
(blum '(trace-moi 20))
(INI 319)
```

Voir l'annexe 1 pour plus de détails.

3°) solution 3

Théorème Il existe une machine à 0 arguments qui s'arrête et donne une *description (indirecte)* de l'entièreté de sa propre trace. Plus précisément, il existe un programme e tel que

$$\phi_e() = d \text{ et } \phi_d() = \text{trace } ((e)),$$

dit autrement, on remplace l'impossible

$$\phi_e() = (e) = \text{trace complète de e}$$

par

$$\phi_{\phi_e()}() = ((e)) = \text{trace complète de (e)}$$

De plus, si on borne la trace, le résultat est accessible aux écoles du dedans :

$$\phi_e() = d \text{ et } \phi_d() = \text{trac-exec } ((e)\ N)$$

avec N assez grand. La solution naturelle est de construire un programme qui donne la description de la *forme* (lambda () (trac x), et de lui appliquer l'opérateur de Kleene k. On obtient un programme e qui génère un programme (lambda () (trac (e))

```
(def 'descrip-trac '(lambda (x)
        (list 'lambda nil (list 'trac (list 'quote (list x))))))

(def 'D-T-Moi (k descrip-trac))
```

On a bien ((D-T-moi)) = (trac '(D-T-moi)) à la première étape près[14]. Voir l'annexe 1.

remarques : 1) la notion d'introspection est intensionnelle et dépend du niveau d'observation. Un homme ne peut pas, seul, étudier sa trace, -au niveau des neurones disons-, jusqu'à son état présent. A ce moment, la régression infinie que Descartes pensait discerner dans la simple reproduction se met effectivement en branle. Cela est reflété par le comportement des programmes décrits plus haut.

2) La solution 3 permet de modéliser une forme d'"instrospection" (au sens large) nécessaire pour monter dans le translateur. Plus précisément il s'agit de capturer, non seulement le code relatif de la machine à laquelle on s'identifie (au niveau adéquat) avec MEC-IND, mais le dernier état instantané de cette machine :

$\phi\phi_e()()$ = ((e)) = (cond ((translaté?) (last (trace complète de (e))
((< >) etc.))

3) Exercice : écrire un programme qui observe sa propre trace, et faire en sorte que celle-ci soit infinie et non périodique. (Hint : écrire avec 2-REC un programme qui observe des segments initiaux (finis) de sa propre trace et change de comportement s'il découvre une période (avec PPTR, voir annexe 1). Bien sûr un tel programme ne découvrira jamais de périodes, c'est pourquoi sa trace sera infinie.

2.2.8 Machines autoréférentiellement correctes

Tout ce que nous avons prouvé peut être formalisé. L'intérêt, dans notre contexte, n'est pas le gain en rigueur, mais le fait que les théorèmes sont accessibles mécaniquement. Nous nous intéressons à ce qu'une machine peut correctement prouver concernant elle-même. Le mot machine est pris dans le sens le plus général du terme : quelque chose de finiment et localement codable relativement à un environnement universel. La thèse de Church justifie l'usage d'une telle définition.

J'adopte un point de vue extérieur où le comportement de la machine, ainsi que la relation entre sa constitution et ce comportement est descriptible en termes impersonnels et extensionnels.

Pour la capacité personnelle, c'est l'opérateur de Kleene k qui permet de procéder, sans crainte de régression infinie, pour *désindexicaliser* la première personne. Pour l'autoréférence correcte, a priori, l'usage de k n'est pas nécessaire. Par exemple, un *altimètre* correct d'un avion indique,

---

[14] c'est un bug, trac, tel qu'il est écrit (voir annexe 1) saute la première étape.

notamment, sa propre hauteur. Il est autoréférentiellement correct si la hauteur qu'il indique est sa propre hauteur (ce qui augmente sa probabilité de "survie"[15]). Le point capital est que l'usage de k garanti l'autoréférence correcte sous réserve de la consistance (dans un sens large) de la machine universelle sous-jaçente et sous-réserve de la pertinence du choix de niveau de description.

Définition Une machine M qui admet une description ou une forme ⌜M⌝ respectivement à un environnement universel *u* **communique de façon autoréférentiellement correcte une proposition p(x)** si p(M) est (matériellement) vrai relativement à *u*.

Matériellement signifie qu'on use de l'*implication matérielle*. Aucune demande de constructivité ou de "causalité" entre l'énoncé de p par M et l'activité de M n'est exigée a priori. Exemple : le programme (lambda (x) (print 0)) est autoréférentiellement correct pour la question "combien de "goto" possèdes-tu dans ton code ?", relativement à un environnement LISP. De même la machine *miroir* est autoréférentiellement correcte pour ses entrées relativement au langage choisi pour l'implémenter.

On approfondira ultérieurement la relation entre machine et système formel, entre le formalisable et le mécanisable, entre machine universelle et environnement. A présent, je justifie le caractère formalisable du théorème de récursion.

Définition Une machine M est **adéquate** si les fonctions partielles calculables ϕ_i sont représentables dans son langage au moyen de termes que je désigne par $\underline{\phi}_i$. De même $\underline{n}$ représente un nom de n dans le langage de la machine.

Définition Une machine est Σ_1-complète si, lorsque $\exists y\ \phi_i(x)=y$, alors la machine est capable de justifier formellement ce fait :

$$\forall i \forall x \forall y \exists y\ \phi_i(x)=y \Rightarrow M \vdash \exists y(\underline{\phi}_i(\underline{x}) = \underline{y})$$

Le caractère formel de la justification permet le caractère vérifiable en un temps fini. Les preuves sont pensées comme des preuves formalisées dans une arithmétique intuitioniste ou classique du premier ordre représentable dans le langage de la machine.

Je dis qu'une formule est Σ_1 si elle est de la forme $\exists x A(x)$ avec A désignant un prédicat[16] total caculable. Dans ce cas, cette machine est Σ_1-complète si :

---

[15] ... surtout à l'atterrissage. J'utilise une version indexicale de la sémantique de Tarski.

[16] B pour BEWEIS (preuve), voir Gödel 1931

p est Σ_1 => (p est vraie => p est prouvable par M)

Comme les ensembles RE sont représentables dans le langage des machines universelles, et que les propositions de la forme $x \in W_y$ sont typiquement Σ_1, une machine universelle peut être considérée[17] comme étant typiquement Σ_1-complète : la preuve de "$\underline{\phi_i}(\underline{x}) = \underline{y}$", (au cas où $\phi_i(x) = y$) est donnée par le calcul de la forme normale U(μjT(i, x, j)). Une *telle* preuve satisfera autant un intuitioniste qu'un *logicien classique*.

Une machine universelle peut prouver au moins tout résultat auquel peut aboutir une machine digitale quelconque (on utilise continuellement la thèse de Church, cela va sans dire), et notamment une *autre* machine universelle (exemple l'arithmétique, l'arithmétique de Robinson, Prolog).

Une telle machine calcule toutes les fonctions totales calculables, (lesquelles sont chacunes représentables, même si on ne sait pas mécaniquement distinguer (prouvablement) leurs représentations de celles des fonctions partielles).

En particulier les opérations de *substitution* sont représentables. Je considère la substitution particulière qui substitue la première variable libre x dans la description ⌜A(x, y)⌝ d'une formule A(x, y) par la description d'un nombre naturel $\underline{n}$:

$$subst(\ulcorner A(x, y) \urcorner, \underline{n}) = \ulcorner A(\underline{n},y) \urcorner$$

Le lemme de diagonalisation est la formalisation (dans le langage d'une machine Σ_1-complète) d'une version particulière du second théorème de récursion.

La représentation de *subst* va être utilisée pour représenter la machine paramétrisatrice *s* utilisée dans le second théorème de récursion.

Le lemme de diagonalisation s'énonce comme suit ($\vdash X$ est abrège M $\vdash$ X) :

Pour toute formule A(x) il existe un énoncé (formule sans variable libre) B telle que $\vdash$ B <-> A(⌜B⌝)[18].

preuve La preuve est semblable à un raisonnement que nous avons déjà fait maintes fois. Considérons la formule *quelconque* A(x), et considérons la formule

---

[17] Ici je sacrifie la rigueur pour ne pas alourdir l'exposé. Voir Smorynski 1985 pour une "arithmétisation" rigoureuse et pertinente pour le présent contexte. Mon exposition repose sur une intuition qui manquait (forcément) à Gödel 1931, celle de l'informatique concrète.

[18] Une version plus générale : Pour toute formule A(x,y) il existe une formule B(y) telle que $\vdash$ B(y) -> A(⌜B(y)⌝,y). La version la plus générale (avec récursion croisée comme dans la planaire) est donnée par Montague 1962, voir aussi Smorynski 1981, Boolos 1979 et Smullyan 1961.

A(subst(x,x)) première diagonalisation

soit $\underline{r}$ = ⌜A(subst(x,x))⌝ , alors l'énoncé :

A(subst($\underline{r}$,$\underline{r}$)) deuxième diagonalisation

est la formule recherchée En effet

⊢ A(subst($\underline{r}$,$\underline{r}$)) <-> A(subst($\underline{r}$,$\underline{r}$))
⊢ A(subst($\underline{r}$,$\underline{r}$)) <-> A(subst(⌜A(subst(x,x))⌝,$\underline{r}$))
⊢ A(subst($\underline{r}$,$\underline{r}$)) <-> A(⌜A(subst($\underline{r}$,$\underline{r}$))⌝)

si B désigne l'énoncé A(subst ($\underline{r}$,$\underline{r}$)), on a

⊢ B <-> A(⌜B⌝)

où ⌜B⌝ désigne un nom ou une description de ce qui est désigné par B. Je commettrai différentes sortes d'abus de notation de ce genre.

Conséquences :

1°) Le premier théorème d'incomplétude (Gödel 1931)

Je rappelle qu'une machine qui admet une description ou une forme M respectivement à un environnement universel *u* **communique de façon autoréférentiellement correcte une proposition p(x)** si p(M) est (matériellement) vrai relativement à *u*.

Définition M (ou la communication de M) est dite auto-référentiellement convainquante si M possède une capacité personnelle par rapport à la prouvabilité formelle relativement à *u* (pour une définition relatée voir Smullyan 1985). L'environnement *u* ne joue ici pas d'autre rôle que celui de processeur pour l'activité de la machine.

L'ensemble des théorèmes de cette machine-théorie ou de cette machine *communicante* est récursivement énumérable. On peut tester mécaniquement si une preuve d'une proposition p donnée par la machine est bien une preuve : on dispose donc d'un prédicat de prouvabilité total calculable[19] : bew(x,y) <-> y est le *nombre de Gödel* (je veux dire une description dans le langage de M) d'une preuve de la proposition codée ou décrite par x.

Dans ce cas on peut définir le prédicat de prouvabilité formel B(x)

[19] Un prédicat est total calculable si la fonction caractéristique du prédicat est totale calculable.

$$B(x) <-> \exists y\ bew(x,y)$$

Comme M est Σ_1-complète, on a

$$M \vdash p => M \vdash B(\ulcorner p \urcorner)$$

Définition M est *saine* si la réciproque est correcte, c-à-d si

$$M \vdash B(\ulcorner p \urcorner) => M \vdash p$$

Je rappelle que M est consistante si $M \nvdash \bot$, et p est indécidable pour M si $M \nvdash p$ et $M \nvdash \neg p$.

Théorème (Gödel 1931) : Pour toute machine saine, consistante Σ_1-complète, il existe une proposition I qui est indécidable pour M[20].

preuve il suffit d'appliquer le lemme de diagonalisation sur la formule $\neg B(x)$. Dans ce cas il existe un énoncé I tel que $M \vdash I <-> \neg B(\ulcorner I \urcorner)$.

A présent $M \vdash I$ entraîne $M \vdash \neg B(\ulcorner I \urcorner)$, mais aussi, par (*), $M \vdash B(\ulcorner I \urcorner)$, et donc $M \vdash \bot$. (absurde puisque la machine est consistante).

De même $M \vdash \neg I$ entraîne $M \vdash B(\ulcorner I \urcorner)$, mais aussi, puisque la machine est supposée saine (**), $M \vdash I$, et alors aussi $M \vdash \bot$.

Remarquons que la preuve est constructive, à la différence de la démonstration donnée dans la section précédente.

En terme de théorie axiomatisable, on voit que si elle est consistante, saine, et assez riche, elle est non seulement incomplète, mais constructivement incomplète. A partir d'une présentation convenable[21] de la théorie , on peut construire la proposition indécidable[22].

---

[20] Gödel supposait que M est ω-consistante plutôt que saine. M est ω-consistante lorsque, s'il n'existe pas de formule A(x) (dans le langage de la machine) telle M prouve $\exists x A(x)$ et en même temps M prouve $\neg A(\underline{n})$ pour chaque n. Si M est ω-consistante M est saine. En effet $M \vdash B(p) => M \vdash \exists x\ Bew(\ulcorner p \urcorner, x)$, donc (par ω-consistance) il existe un n tel que $M \nvdash -Bew(\ulcorner p \urcorner, \underline{n})$, mais Bew est prouvablement (par M) totale calculable (voir *infra*), donc $M \vdash B(\ulcorner p \urcorner, \underline{n})$ et on peut extraire de n une preuve de p, donc $M \vdash p$. Notons que la consistance n'entraîne pas l'ω-consistance. En introduisant un prédicat de prouvabilité R(x) un peu plus sophistiqué :

$\exists y\ (Bew(y,x) \& \forall z\ (z<y-> \neg Bew(z, \ulcorner \neg x \urcorner)))$

Rosser 1936 montre l'existence d'une proposition indécidable pour toute théorie simplement consistante. Je mentionne aussi que la "machine" considérée par Gödel était la théorie PRINCIPIA MATHEMATICA de Russell & Whitehead 1910.

[21] Cela veut dire qu'à partir de la présentation de T, on peut extraire le prédicat de prouvabilité Bew(x, y) et prouver qu'il est total calculable. Ex : la théorie qui admet un certain axiome sous réserve que Fermat est vrai, n'est pas convenablement présentée.

[22] Il n'est pas difficile de montrer que l'ensemble des propositions prouvables est un ensemble créatif.

2°) Théorème de Tarski

L'énoncé I que Gödel a construit affirme sa propre *improuvabilité* par la machine M. Peut-on construire un énoncé qui, à la façon d'Epiménide le crétois, affirme sa propre *fausseté* ? Ce serait sans aucun doute le cas s'il existait un prédicat de vérité V(x) représentable dans le langage de M :

$$M \vdash A <-> V(\ulcorner A \urcorner)$$

car on pourrait alors appliquer le lemme de diagonalisation au prédicat $\neg V(x)$. Il existerait alors un énoncé E tel que

$$M \vdash E <-> \neg V(\ulcorner E \urcorner) <-> \neg E,$$

ce qui entraînerait, illico, l'inconsistance de M.

3°) M reflète le modus ponens

(Sketch) Si $B(\ulcorner p \urcorner)$ & $B(\ulcorner p->q \urcorner)$ est vraie, alors $B(\ulcorner p \urcorner)$ et $B(\ulcorner p->q \urcorner)$ sont vraies et, la théorie étant saine on a $Bew(\ulcorner p \urcorner,x1)$ et $Bew(\ulcorner p->q \urcorner,x2)$. Dans ce cas, il est facile de construire (algorithmiquement à partir de x1 et x2) une description d'une preuve de q, que T sait produire par Σ_1-complétude. Donc $M \vdash B(\ulcorner q \urcorner)$. On a donc, par le théorème de déduction :

$$M \vdash B(\ulcorner p \urcorner) \& B(\ulcorner p->q \urcorner) -> B(\ulcorner q \urcorner)$$

M formalise ainsi le modus ponens.

4°) Enoncés de Henkin et machine de Löb

L'énoncé de Gödel affirmant sa propre non-prouvabilité dans T est d'office aussi vrai que non prouvable, si T est consistante. Qu'en est-il d'un énoncé affirmant sa propre prouvabilité. Cet énoncé existe, avec le lemme de diagonalisation et le fait que B est représentable :

$$T \vdash H <-> B(\ulcorner H \urcorner)$$

Cet énoncé, appelé *énoncé de Henkin*, peut a priori être autant vrai (et alors prouvable) que faux (et alors improuvable si M est consistante).

Henkin a posé cette question en 1952, et Löb y répondit en 1955 en montrant que pour une vaste classe de théories les propositions de Henkin sont *vraies* et *prouvables !*

Les théories concernées sont celles qui non seulement sont Σ_1-complète, comme l'arithmétique de Robinson, mais sont capables ***de prouver leur propre Σ_1-complétude***, comme l'arithmétique de Heiting, ou celle de Peano. C'est-à-dire, que non seulement on a, pour p, formule Σ_1.

$$M \vdash p \Rightarrow (p \Rightarrow M \vdash B(\ulcorner p \urcorner)),$$

mais on a, toujours pour une formule p qui est Σ_1,

$$M \vdash p \rightarrow B(\ulcorner p \urcorner)$$

Pour les formules Σ_1, M prouve que si elles sont vraies alors elles sont prouvables. De telles machines prouvent leur propre Σ_1-complétude. D'une certaine façon, elles sont à même de communiquer leur propre (Turing)-universalité.

En particulier comme B(x) est Σ_1, M prouve

$$M \vdash B(\ulcorner p \urcorner) \rightarrow B^{\ulcorner}(B(\ulcorner p \urcorner)^{\urcorner})$$

Elliptiquement, une telle machine (théorie) est non seulement universelle, mais elle reflète déductivement son universalité (au moins potentielle).

L'inférence $M \vdash p \Rightarrow M \vdash B(\ulcorner p \urcorner)$ confère un premier degré d'introspection pour un démonstrateur de théorème et de métathéorème le concernant. Il signifie que si M prouve p, M sait prouver que M prouve p.

La proposition $B(\ulcorner p \urcorner) \rightarrow B^{\ulcorner}(B(\ulcorner p \urcorner)^{\urcorner})$, prouvable par M, confère à M un deuxième degré d'introspection reflétant le degré précédent. Il signifie que M sait prouver que si elle prouve p, elle peut prouver qu'elle peut prouver p.

résumé[23]:

1) $M \vdash p \Rightarrow M \vdash B(\ulcorner p \urcorner)$	(L1)
2) $M \vdash B(\ulcorner p \urcorner) \& B(\ulcorner p \rightarrow q \urcorner) \rightarrow B(\ulcorner q \urcorner)$	(L2)
3) $M \vdash B(\ulcorner p \urcorner) \rightarrow B^{\ulcorner}(B(\ulcorner p \urcorner)^{\urcorner})$	(L3)

Notons que 2) est équivalente à

2') $M \vdash B(\ulcorner p \rightarrow q \urcorner) \rightarrow (B(\ulcorner p \urcorner) \rightarrow B(\ulcorner q \urcorner)$	(L2')

On remarquera, qu'en clignant un peu des yeux, on reconnaît les axiomes du système modal normal K4, c'est-à-dire un système axiomatisé par le calcul propositionnel classique étendu par

1) $M \vdash p \Rightarrow M \vdash \Box p$	(L1)
2) $M \vdash \Box p \& \Box(p \rightarrow q) \rightarrow \Box q$	(L2)
3) $M \vdash \Box p \rightarrow \Box\Box p$	(L3)

[23] C'est une version moderne des conditions de dérivabilité de Hilbert et Bernays, elles sont explicitées sous cette forme par Löb (1955).

où, informellement, □p correspond à B(⌜p⌝). Ceci sera précisé et utilisé dans la section suivante.

Le théorème de Löb (1955)

$$M \vdash B(\ulcorner p \urcorner) \rightarrow p \Rightarrow M \vdash p$$

Le théorème de Löb donne la solution au problème posé par Henkin. Pour les formules h qui affirment leur propre prouvabilité par M (relativement à *u*), c'est-à-dire M ⊢ h<->B(⌜h⌝), qui existe avec le lemme de diagonalisation, on a en particulier M ⊢ B(⌜h⌝)->h, ce qui entraîne, par Löb, M ⊢ h. h est donc prouvable par M. Elle est vraie aussi, que M soit consistante ou inconsistante.

Ce résultat est assez extraordinaire, on aurait pu croire M capable de prouver B(⌜p⌝)->p, quel que soit p. Au lieu de cela, M prouve B(⌜p⌝)->p seulement si M prouve p. Les machines saines, consistantes et suffisamment riches que pour prouver leur propre Σ_1-complétude font preuve d'une modestie inattendue. En particulier, si M est consistante, M ⊬ ⊥, et par Löb, M ⊬ B(⌜⊥⌝)->⊥, c'est-à-dire M ⊬ ¬B(⌜⊥⌝), autrement dit encore M ne prouve pas sa consistance. Le théorème de Löb permet une dérivation facile du *second théorème d'incomplétude* de Gödel.

Löb suggère que la démonstration qu'il donne de son théorème apporte une nouvelle dérivation paradoxale dans les langues naturelles. La dérivation ne comporte pas de négation à la différence de la plupart des dérivations paradoxales traditionnelles[24]. Il s'agit d'une preuve de *p* où *p* est votre proposition préférée, par exemple l'existence de Saint Nicolas (Boolos 1979).

Paradoxe : Saint Nicolas existe !,
En effet considérons l'énoncé

si cet énoncé est vrai alors Saint-Nicolas existe (*)

Premier temps. Je vais d'abord démontré que cet énoncé, c-à-d (*), est vrai. Supposons que cet énoncé soit vrai (c'est la prémisse de (*)), alors (*) et sa prémisse sont vraies, *dans ce cas* Saint Nicolas existe. J'ai montré que si la prémisse de (*) est vraie, la conclusion de (*) est vraie. J'ai donc montré que (*) est vraie.

[24] A l'exception d'un paradoxe dû à Curry 1942. (voir aussi Rosser 1955).

Deuxième temps. Nous savons que (*) est vraie. Mais la prémisse de (*) affirme justement cela. Donc la prémisse de (*) est vraie. Comme (*) et sa prémisse sont vraie, la conclusion de (*) est vraie aussi. Laquelle conclusion affirme l'existence de Saint Nicolas.

Ce paradoxe n'est pas reproductible dans le langage de M puisque, *par Tarski*, M n'a pas de prédicat de vérité. Mais M possède une approximation de la vérité, la prouvabilité. L'énoncé considéré par Löb pour démontrer son théorème est l'énoncé (*), avec un prédicat de prouvabilité :

si cet énoncé est prouvable alors p (**)

ou p est une proposition quelconque expressible par M., y compris ∃x(x=Saint Nicolas) si le langage de M possède un terme Saint Nicolas pour désigner Saint Nicolas.

preuve de Löb

Avec le lemme de diagonalisation il existe, pour toute proposition p, un énoncé q de type (**) prouvable par M :

$$M \vdash q <-> (B(\ulcorner q \urcorner)->p)$$

En particulier M ⊢ q -> (B(⌜q⌝)->p).
Par L1 M ⊢ B(⌜q -> (B(⌜q⌝)->p⌝)) (1)
Par L2' M ⊢ B(⌜q -> (B(⌜q⌝)->p⌝)) -> (B(⌜q⌝)->B(B(⌜q⌝)->p⌝)) (2)
Par (1) et (2) avec le modus ponens :
M ⊢ B(⌜q⌝)->B(B(⌜q⌝)->p⌝) (3)
par L2' encore M ⊢ B(⌜q⌝)->(B(B(⌜q⌝))->B(⌜p⌝)) (4)
Par L3 M ⊢ B(⌜q⌝)->B(⌜p⌝)) (5)

A présent, la prémisse du théorème de Löb est M ⊢ B(⌜p⌝)->p, quel que soit p. Donc avec M ⊢ q <-> (B(⌜q⌝)->p), qui
entraîne (5), la prémisse de Löb entraîne M ⊢ B(⌜q⌝)->p (6)
mais si q <-> (B(⌜q⌝)->p), on a grâce à 6), M ⊢ q. (7)
Par L1 et 7), on a M ⊢ B(⌜q⌝), et avec 6), M ⊢ p.

Löb a démontré un beau théorème concernant l'arithmétique de Peano (PA : l'Escherichia Coli des machines saines consistantes, et sachant prouver leur propre Σ_1-complétude). Ces machines savent en fait prouver elles-mêmes le "théorème de Löb" au sujet de leur propre prouvabilité. Dit autrement le raisonnement de Löb est formalisable dans le langage de la machine (voir Boolos 1979, voir 2.3.3) : quel que soit l'énoncé p,

$$M \vdash B(\ulcorner B(\ulcorner p \urcorner) \to p \urcorner) \to B(\ulcorner p \urcorner).$$

En particulier à nouveau, $M \vdash B(\ulcorner B(\ulcorner \bot \urcorner) \to \bot \urcorner) \to B(\ulcorner \bot \urcorner)$, et donc

$$M \vdash B(\ulcorner \neg B(\ulcorner \bot \urcorner) \urcorner) \to B(\ulcorner \bot \urcorner)$$

Désignons la consistance $\neg B(\ulcorner \bot \urcorner)$, telle qu'elle s'exprime naturellement pour M, par CON, on a

$$M \vdash B(\ulcorner CON \urcorner) \to \neg CON$$

Ce qui fait de la consistance, une (lointaine ?) cousine de la conscience (LWV). Ce qu'on avait déjà aperçu sur les diagrammes réalistes de la translation. De plus amples et précises informations sont proposées en 2.3.3.

5°) le virus (Hofstadter 1979, Solovay 1985)

Hofstadter compare les énoncés de Henkin aux virus. Il introduit une notion d'énoncé explicite de Henkin. Un tel énoncé E non seulement affirme l'existence d'une preuve de lui-même E, mais l'exhibe explicitement. A la différence des énoncés (implicites) de Henkin, qui, par Löb, sont toujours vrais et prouvables, il est facile de construire un énoncé explicite qui est faux. En effet avec le lemme de diagonalisation on peut construire un énoncé p qui dit que 733, ou n'importe quoi qui n'est pas (une description d') une preuve de p, est une preuve de lui-même :

$$p \leftrightarrow Bew(\ulcorner p \urcorner, 733).$$

Existe-il un énoncé explicite de Henkin qui soit vrai (et donc prouvable). Un tel énoncé est l'analogue de programme *complètement* introspectif, c-à-d ceux qui donnent une description (nécessairement indirecte) de leur propre trace complète (voir plus haut). La notion de trace correspondant ici à la notion de preuve. En utilisant le fait que Bew et la paramétrisation peuvent appartenir à une énumération *du dedans* [25] Solovay 1985 démontre en détail l'existence d'énoncés explicites de Henkin qui sont vrais.

6°) Les machines de Rogers (1967)

Un énoncé p est consistant pour une théorie T, si $T \nvdash \neg p$

Que peut-on dire d'un énoncé (vu comme une théorie, ou une machine) qui affirme, relativement à T, sa propre consistance :

$$T \vdash p \leftrightarrow \neg B(\ulcorner \neg p \urcorner)$$

[25] Voir aussi Royer 1987 et Selman 1990.

On a en particulier T ⊢ B(⌜¬p⌝) -> ¬p, cela entraîne par Löb, T ⊢ ¬p. Et comme T est consistante , cela signifie que p est faux. Un énoncé "assez riche" qui affirme sa *propre* consistance est faux.

Remarque Il s'agit de bien distinguer un énoncé de consistance, et un énoncé de d'autoconsistance. Prenons une théorie T_{123} consistante, saine Σ_1-complète. Elle a une forme du genre {1, 2, 3} avec :

1 = axiome 1
2 = axiome 2
3 = axiome 3

La proposition du second théorème d'incomplétude de Gödel con ({1, 2, 3}) est indécidable dans la théorie. Par le théorème de déduction de Herbrand, les deux extensions suivantes sont consistantes :

1 = axiome 1	1 = axiome 1
2 = axiome 2	2 = axiome 2
3 = axiome 3	3 = axiome 3
4 = con {(1, 2, 3})	4 = ¬con {(1, 2, 3})

L'énoncé de Rogers est plutôt équivalent à une théorie du style :

1 = axiome 1
2 = axiome 2
3 = axiome 3
4 = con {(1, 2, 3, 4})

Il affirme que la théorie $T_{1234} = T_{123}$ + *moi-même* est consistante. On peut dériver l'inconsistance d'une telle théorie directement à partir du second théorème de Gödel. Si une telle théorie était consistante, alors elle prouverait qu'elle est consistante (les axiomes sont formellement prouvables par des preuves de longueur 1), ce qui contredit le second théorème de Gödel. L'axiome 4 est faux, et T_{1234} prouve le faux.

2.2.9 Complexité (Φ_i,W_i)

J'introduis à présent une notion fondamentale de l'informatique théorique qui mesure la complexité des éxécutions. Cette notion a un certain rôle dans 2 3. Pour l'instant elle va permettre d'illustrer quelques diagonalisations de plus, ainsi que l'usage de quantificateurs particuliers pour les voisinages de l'infini.

1°) Mesure de complexité de Blum

$\{\phi_i \mid i \in \omega\}$ désigne une numérotation acceptable du dehors (de l'ensemble des fonctions partielles calculables).

On peut associer à chaque fonction ϕ_i une fonction β_i, qui sur l'entrée x donne le nombre d'étape de calcul de $\phi_i(x)$ si $\phi_i(x)$ est définie, et est indéterminée si $\phi_i(x)$ est indéterminée.

Cette association peut se faire de façon uniforme. Il suffit de construire une fonction universelle reposant sur l'itération, et de compter simultanément le nombre d'étapes de calcul. C'est ce que fait par exemple **ini-st-n-muette** qui calcule simultanément ϕ_i et β_i (voir annexe 1).

```
? (ini-st-n-muette)
(fact 3)
(RESULTAT  INI-ST-N-MUETTE  6)

24              ;  =  βi avec i = ⌜fact⌝

(fact-iter 3)
(RESULTAT  INI-ST-N-MUETTE  6)

43              ;  =  βi avec i = ⌜fact-iter⌝

(fact-an 3)
(RESULTAT  INI-ST-N-MUETTE  6)

36              ;  =  βi avec i = ⌜fact-an⌝
```

où fact, fact-an, fact-iter sont trois programmes qui calculent la factorielle. fact correspond à la définition récursive avec l'appel du nom. Fact-an est la version anonyme produite par (k f-fact) (voir plus haut), et Fact-iter est une version FORTRAN-like, c-à-d purement itérative.

On trouvera aussi dans l'annexe λxBLUM(x) et λxyBLUM-EXEC(x,y) qui calcule respectivement β_i sur une forme x = (i data) et le prédicat B décrit ci-dessous (appelé BLUM-EXEC dans l'annexe 1).

$\beta_i(x)$ mesure la complexité du calcul $\phi_i(x)$, et β_i mesure la complexité de ϕ_i.

Les fonctions β_i vérifient deux propriétés importantes :

1) $dom(\beta_i) = dom(\phi_i)$
2) le prédicat B(i,x,y) définit par $\beta_i(x) = y$ est total calculable.

Remarquons que les prédicats définis par $\beta_i(x) \geq y$, ou par $\beta_i(x) \leq y$ sont eux-mêmes totals calculables. On peut les tester avec BLUM-EXEC (voir annexe).

Blum définit une mesure de complexité par la donnée d'une suite de fonctions (Φ_i) respectant 1) et 2), appelés axiomes de Blum.

(ϕ_i,Φ_i) est parfois appelé *espace* de Blum (Calude 1988, Blum&Al. 1973).

Je limiterai les raisonnements sur la mesure de complexité concrète (β_i), mais je n'utiliserai pas autre chose que les axiomes de Blum. En fait les mesures de Blum vérifient toutes sortes de propriétés agréables qui permettent de les identifier dans les voisinages de l'infini.

Je rappelle qu'un voisinage de zéro est un ensemble de la forme $\{x \mid x < n\}$ et un voisinage de l'infini est de la forme $\{x \mid x > n\}$.

Deux quantificateurs spéciaux sont utiles dans ce contexte. $\forall^{\infty}x$, qui est lu "pour presque tous les x". $\forall^{\infty}x\ p(x)$ signifie que $p(x)$ est vrai partout à un nombre fini d'exception près. Notons que $\forall^{\infty}x\ p(x)$ est équivalent à *il existe un voisinage de l'infini où p est vrai*. $\exists^{\infty}x\ p(x)$ est , à l'instar des quantificateurs habituels, ou des opérateurs modaux, définissable par

$$\neg \forall^{\infty}x \neg\ p(x),$$

c-à-d qu'on n'a pas presque partout $\neg p(x)$. Ce qui revient à dire *il existe une infinité de x telle que p(x)*.

Le fait que l'on travaille dans les voisinages de l'infini, c-à-d à terme indéterminé, peut éloigner ceux qui ne visent que des projets à terme déterminé (dans les voisinages de zéro). L'usage des voisinages de l'infini exemplifie le fait que les machines universelles jouent des jeux infinis contrairement à ce que prétend John Carse 1986. Comme John Carse défend l'idée que le *sens de la vie* est lié au fait que nous jouons à des jeux infinis (thèse de Carse), cela devrait être une bonne nouvelle puisque cela signifie que les machines y participent. Ceci est bien sûr d'office impliqué par la thèse de Carse + MDI. En ce sens, les théorèmes qui vont suivre confirment MDI pour ceux qui admettent la thèse de Carse.

Le fait que P soit fermé pour la diagonalisation entraîne l'existence de fonctions, comme $g(x) = \phi_x(x)+1$, nécessairement partielles (en particulier $g(\ulcorner g \urcorner)$ diverge[26])

Peut-on utiliser la diagonalisation pour prouver l'existence d'une fonction ou d'un prédicat total(e) calculable *arbitrairement complexe* ?

La réponse est affirmative. Comment ?

---

[26] Je rappelle que $g(\ulcorner g \urcorner) = \phi_r(r)$, si r est l'indice de g.

On voudrait démontrer l'existence d'une fonction totale arbitrairement complexe, c-à-d une fonction ϕ_e telle que

$$\forall x\ \beta_e(x) > h(x)$$

avec h(x) une fonction récursive aussi croissante qu'on le désire. Toutefois, il est possible d'accélérer arbitrairement $\phi_e(x)$ dans les voisinages de zéro, en donnant une table des résultats "pré-calculés", on ne pourra, dès lors, démontrer le caractère arbitrairement complexe de ϕ_i seulement dans un voisinage de l'infini :

$$\forall^{\infty} x\ \beta_i(x) > h(x)$$

2°) Théorème de Rabin

Je propose quatre résultats, le dernier est le plus général et il suffirait, mais mon intention est d'illustrer l'usage de la (double) diagonalisation dans le contexte de la complexité[27].

Théorèmes Il existe une fonction telle que, quelle que soit la machine (programme, code, index) qui la calcule, le nombre d'étapes du calcul sera arbitrairement grand sur

1) au moins une entrée ;

2) sur au moins une infinité d'entrées (sur une $\exists^{\infty} x$) ;

3) sur presque toutes les entrées (sur une $\forall^{\infty} x$) ;

4) idem que 3), mais avec une fonction à valeur dans {0,1}, c-à-d un prédicat.

1) $\forall N\ \exists g\ \ g = \phi_e \rightarrow \exists x\ \beta_e(x) > N$ (*[28])

preuve considérons la fonction

$$g(x) = \phi_x(x)+1 \text{ si } \beta_x(x) =< N$$
$$= 733 \text{ sinon}[29]$$

g est totale calculable puisque $\beta_x(x) =< N$ est décidable.

---

[27] Ces résultats figurent parmi ce que Hartmanis considère comme devant être connus par l'étudiant informaticien. J'y incluerais l'étudiant philosophe, afin de ne pas exclure des philosophies, qui, comme la philosophie mécaniste, repose, pour des raisons philosophiques, sur un minimum de connaissances mathématiques.

[28] Pour ne pas alourdir les notations, je quantifie sur une fonction totale ($\forall g$), mais il est encore possible de ne quantifier que sur les nombres naturels et nos propositions sont encore des propositions de l'arithmétique. Par exemple ici il faudrait écrire

$\forall N\ \exists i\ \forall k\ (\phi_k=\phi_i) \rightarrow \exists x\ \beta_k(x) > N)$, en remplaçant encore $\phi_k=\phi_i$ par sa définition arithmétique avec le prédicat de Kleene.

[29] ... ou n'importe quoi de récursif.

on a : si $g = \phi_e$ alors $\beta_e(e) > N$. En effet si $\beta_e(e) =< N$, par définition de g, $g(e) = \phi_e(e) = \phi_e(e)+1$, contradiction[30].

2) $\forall h \exists g \;\; g = \phi_e \rightarrow \exists^{\infty} x \; \beta_e(x) > h(x)$
Preuve Considérons la suite

(0,1,0,1,2,0,1,2,3,0,1,2,3,4,0,1,2,3,4,5,0 ...)

et appelons r, la fonction récursive qui calcule cette suite.
Considérons la fonction g définie ainsi

$$g(x) = \begin{cases} \phi_{r(x)}(x)+1 \text{ si } \beta_{r(x)}(x) =< h(x) \\ 733 \text{ sinon} \end{cases}$$

g est totale calculable et il existe un e tel que $g = \phi_e$. Vu la définition de r, il existe une infinité de x tel que $r(x)=e : \exists^{\infty} x \; r(x) =e$. Pour ces valeurs on a $\beta_{r(x)}(x) > h(x)$. En effet, si ce n'était pas le cas, il existerait un x_o avec $r(x_o) = e$ et $\beta_{r(x_o)}(x_o) =< h(x_o)$. Dans ce cas $g(x_o) = \phi_{r(x_o)}(x_o)+1$, mais $r(x_o) = e$ et $g = \phi_e$, donc $g(x_o) = \phi_{r(x_o)}(x_o)$. Contradiction.

3) $\forall h \exists g \;\; g = \phi_e \rightarrow \forall^{\infty} x \; \beta_e(x) > h(x)$
preuve Pour chaque x définissons l'ensemble

$$I_x = \{i \mid i =< x \;\&\; \beta_i(x) =< h(x)\}$$

Considérons la fonction

$$g(x) = \begin{cases} 1+\sum_{\text{sur } i \in Ix} \phi_i(x) \text{ si } I_x \text{ n'est pas vide} \\ 733 \text{ sinon} \end{cases}$$

g est récursive (totale calculable) parce que $\beta_i(x) =< h(x)$ est décidable, et il existe e telle que $g = \phi_e$. On a $\forall x \;\; x > e \rightarrow \beta_e(x) > h(x)$. Et donc $\forall^{\infty} x \; \beta_e(x) > h(x)$, quel que soit l'algorithme ou la machine calculant g. En effet, si ce n'était pas le cas, alors

$$\exists x_o \;\; x_o > e \;\&\; \beta_e(x_o) =< h(x_o)$$

dans ce cas $e \in I_{x_o}$, par définition de I_x, mais alors

---

[30] Que pensez-vous de la "réfutation-de-Lucas" suivante : j'ai un algorithme pour calculer g rapidemment sur i, puisque sur i elle vaut 733 !

$$g(x_o) = \phi_e(x_o) = 1+\sum_{\text{sur } i\in I_{x_o}} \phi_i(x_o) = 1+...+\phi_e(x_o)+...$$

Contradiction (on se rappelle que g est totale).

4) Théorème de Rabin $\forall h\ \exists g\in 2^\omega\ \ g = \phi_e \rightarrow \forall^\infty x\ \beta_e(x) > h(x)$

preuve nous utilisons encore les $I_x = \{i \mid i =< x\ \&\ \beta_i(x) =< h(x)\}$. Avec 2-REC, il existe $g = \phi_e$ avec

$$\phi_e(x) = \begin{cases} 1--\phi_i(x) & \text{si i est le plus petit entier dans } I_x \text{ qui n'a jamais été utilisé dans les calculs de } \phi_e(x-1), \phi_e(x-2), ... \phi_e(0). \\ 733 & \text{s'il n'y a pas de tel i.} \end{cases}$$

où x--y = x-y si x > y, et 0 sinon. ϕ_e est total calculable et elle appartient à 2^ω. On a $\forall^\infty x\ \beta_e(x) > h(x)$. En effet, supposons que non. Dans ce cas $\exists^\infty x\ \beta_e(x) =< h(x)$. En particulier $\exists^\infty x\ (x > e)\ \&\ \beta_e(x) =< h(x)$. Il existe donc une suite de nombres a_i, avec $a_i > e$, et $\beta_e(a_i) =< h(a_i)$. Par définition de I_x on a $e \in I_{a_o}$.

e ne peut pas être le plus petit entier jamais utilisé dans les calculs de $\phi_e(a_o-1)$, $\phi_e(a_o-2)$,..., $\phi_e(0)$; sinon $\phi_e(a_o) = 1--\phi_e(a_o)$. Il existe donc, dans I_{a_o}, un plus petit entier qui n'a jamais été utilisé et qui est plus petit que e. Appelons-le e_{a_o}.

A présent le même raisonnement vaut pour tous les a_i, et comme on utilise jamais deux fois le même index, on a $e_{a_o} \neq e_{a1} \neq e_{a2} \neq e_{a3} \neq ...$, il en existe une infinité et ils sont tous plus petits que e. Contradiction.

Remarque : les 4 preuves sont constructives. Elles sont toutes basées sur une double diagonalisation. 2-REC est utilisé dans la 4ième preuve car g est définie par une récursion élémentaire.

3°) Quelques notions et théorèmes supplémentaires

-Les sauts de Borodin

A une fonction récursive[31], ϕ_i,on peut associer la classe C_i des fonctions récursives calculables sur x par une machine en moins de $\phi_i(x)$ étapes presque partout.

---

[31] Sauf mention du contraire, *fonction récursive* signifie ici *fonction totale calculable*.

$$C_i = \{f \mid \exists j\ f = \phi_j \ \&\ \forall^{\infty} x\ \beta_j(x) =< \phi_i(x)\}$$

C_i est la *classe de complexité* déterminée par la fonction ϕ_i.

Comme nous avons démontré l'existence de fonctions arbitrairement complexes, on a que pour chaque classe C_i il existe une classe C_j avec $C_i \subsetneq C_j$.

On a $\forall^{\infty} x\ \phi_a(x) =< \phi_b(x)$ -> $C_a @ C_b$

Quand l'inclusion $C_a @ C_b$ est-elle stricte ? Borodin a démontré l'existence d'une infinité de sauts, arbitrairement grands, dans les échelles de complexité (et ça pour toute mesure de complexité vérifiant les axiomes de Blum).

Théorème Pour toute fonction h, totale calculable, telle que h(x) >= x, il existe une fonction totale calculable f, arbitrairement large (presque partout), telle que :

$$C_f = C_{h \circ f}.$$

Définition Une fonction est f-facile si elle appartient à C, et sinon on dit qu'elle est f-difficile. Le théorème du saut de Borodin est un résultat assez contre-intuitif.

Imaginons $h = \lambda x\ 2^{2^{2^{\cdot^{\cdot^{\cdot^{x}}}}}}$, ou n'importe quelle fonction très croissante, le théorème du saut affirme l'existence d'une classe de complexité C_f telle que les fonctions $\lambda x\ 2^{2^{2^{\cdot^{\cdot^{\cdot^{f(x)}}}}}}$ -facile, sont f-facile. Ou, dit de façon contraposée, les fonctions f-difficiles sont $\lambda x\ 2^{2^{2^{\cdot^{\cdot^{\cdot^{f(x)}}}}}}$ -difficiles. Malgré qu'on augmente super-exponentiellement les ressources (le nombre d'étapes de calcul), on ne sait pas calculer de nouvelles fonctions.

Le temps de calcul, comme le nombre d'étapes, vérifie les axiomes de Blum. Prenons une machine universelle très rapide (pensons au CRAY 1), et une machine universelle très lente (pensons à *vous* imitant la machine de Babbage avec du crayon et du papier). L'existence des sauts implique l'existence de délais pour lesquelles la machine lente calcule la même classe de fonctions que la machine rapide.

Ce résultat montre aussi qu'il n'est pas possible d'élargir *constructivement* des classes de complexité *arbitraire*[32].

théorème d'accélération de Blum

Nous avons une notion de complexité pour les machines, les programmes, les index, les algorithmes. Cette notion de complexité est

---

[32] Il existe un théorème, dit *de compression*, qui montre que si on se limite, par exemple, à des classes de complexité définie par des β_i, il existe une procédure uniforme (constructive) permettant d'élargir la classe. Ceci est rendu possible par le fait que $\beta_i(x) = y$ est décidable à la différence de $\phi_i(x) = y$.

intensionnelle. Peut-on définir une notion de complexité équivalente directement pour les fonctions (au sens extensionnel du terme) ?

Blum a démontré qu'une telle notion n'existe pas. Il a démontré l'existence de fonction qui n'ont pas d'algorithme optimal.

De façon équivalente, il existe des fonctions pour lesquelles il n'existe pas de meilleure machine pour les calculer, ou une machine A est meilleure qu'une machine B, si elle travaille plus vite, ou plus généralement si elle utilise moins de ressources.

Dans l'énoncé du théorème h joue encore le rôle d'une fonction arbitrairement croissante. Par exemple $h = \lambda x\, 2^{2^{2^{\cdot^{\cdot^{\cdot^{x}}}}}}$.

Théorèmes de Blum

a) version constructive

Soit h une fonction totale calculable arbitraire. Il existe une fonction totale calculable f telle que $f = \phi_i \rightarrow \exists j$ tel que :

1) ϕ_j est totale et $\forall^{\infty}\mathbf{x}\ \ \phi_j(x) = \phi_i(x)$;
2) $\forall^{\infty}x\ \ h(\beta_j(x)) < \beta_i(x)$.

Le "il existe" est constructif. A partir de l' index i (quelconque) de f, on peut exhiber l'index j.

ϕ_j & ϕ_i sont presque identiques. Elles sont identiques à l'exception d'un nombre fini d'entrées, elles peuvent différer sur un voisinage de 0.

b) version non constructive

Soit h une fonction totale calculable arbitraire. Il existe une fonction totale calculable f telle que $f = \phi_i \rightarrow \exists j$ tel que :

1) ϕ_j est totale et $\forall\mathbf{x}\ \ \phi_j(x) = \phi_i(x)$;
2) $\forall^{\infty}x\ \ h(\beta_j(x)) < \beta_i(x)$.

A présent ϕ_j & ϕ_i sont strictement identiques, on peut cependant montrer que le "il existe" est, en général, non constructif. j existe, mais il n'est pas possible de l'exhiber mécaniquement à partir de i. (Blum 1967, Blum 1969, Schnorr 1973, Calude 1988 page 250)

En utilisant les W_i à la place des ϕ_i, on peut démontrer des résultats **similaires** pour les fonctions *partielles* calculables.

Définitions

1) un ensemble RE E est dit (*constructivement*) accélérable si, quelle que soit la fonction h -arbitrairement croissante-, et quel que soit l'index i de E (c-à-d $W_i = E$), il existe (*on peut construire à partir de i*) un index j tel que :

$$\exists^{\infty}x \ \beta_i(x) > h(\beta_j(x)).$$

2) Une fonction partielle calculable est (*constructivement*) accélérable si son graphe est (*constructivement*) accélérable.

-Blum et Marquez (1973)

Blum et Marquez ont caractérisé la classe des ensembles RE qui sont constructivement accélérables.

Il ressort de ces travaux, ainsi que de ceux de Van Emde Boas, (voir Calude 1988) que

E *créatif* -> E est *constructivement accélérable*[33]

De façon plus ou moins similaire, on peut montrer que si T est une théorie assez riche, l'adjonction d'une proposition indécidable permet non seulement de décider une infinité de nouvelles propositions, mais aussi de raccourcir les démonstrations (d'accélérer les procédures de preuves) d'une infinité de formules prouvables dans T (voir Gödel 1936, Royer 1989).

Plus grossièrement, cela signifie qu'il existe des tâches pour lesquelles une machine universelle arbitrairement lente, peut être plus rapide qu'une machine universelle arbitrairement rapide, la machine lente étant munie d'un logiciel adéquat. Dans les voisinages de l'infini, un bon logiciel peut ainsi pallier à la lenteur d'une quelconque machine universelle.

Cela montre encore qu'une machine universelle peut toujours progresser dans l'efficacité, celle-ci dérivant d'une mesure de Blum.

Comme le théorème du saut, le résultat de Blum est contre-intuitif. Le sentiment d'impossibilité de la proposition est en grande partie dû au fait que ces résultats seraient contradictoires s'ils portaient sur les voisinages de 0.

Ces résultats ne peuvent pas être directement utilisés en vue des applications pratiques en informatique, celles-ci cherchant plutôt à accélérer les machines dans les voisinages de 0. Mais ils confirment la vision de Lafitte comme quoi les machines peuvent évoluer.

---

[33] Blum & Marques : E subcréatif <-> E est constructivement accélérable (voir def de Salomaa 1985 de subcréatif), + 1°) Pour-El 1970 : une théorie est effectivement extensible si et seulement si elle est effectivement inséparable, et 2°) Royer 1989 : une théorie est effectivement inséparable ssi elle est effectivement accélérable. Donc les théorie effectivement extensibles sont les théories effectivement accélérables. La subcréativité peut être considérée comme une notion de créativité plus générale applicable aux écoles du dedans.

2.2.10 La thèse de Church (suite) et le mécanisme

1°) Remarque sur l'identité Leibnitzienne et l'identité personnelle

On se rappelle de la ritournelle : *si DX donne XX* (première diagonalisation), *DD donne DD*. On peut d'ailleurs écrire cela aisément en LISP :

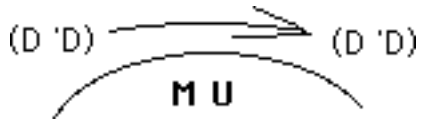
```
? (defun d (x)
   (list x (list 'quote x))
  )
D

? (D 'D)
(D 'D)
```

Lorsqu'on écrit DD = DD, ou (D 'D) = (D 'D) en Lisp, l'égalité est ambiguë. DD = DD, comme DX = DX, comme n'importe quoi = à "lui-même", c'est l'identité Leibnitzienne.

D'autre part DD = DD, de la même façon que (Fact 5) = 120. Cette identité signifie que le terme (Fact 5) dénote 120.

De même le terme DD dénote DD, la forme (D 'D) dénote la forme (D 'D). Les phrases autoréférentielles en langue naturelle, comme "je suis fausse" ne sont autoréférentielles" que relativement à un être humain qui la comprend, et qui comprend.

En ce qui concerne (D 'D), un interpréteur LISP suffit. L'autodénotation qui repose sur une double diagonalisation nécessite un système assez riche que pour pouvoir, comme Q ou PA, représenter la diagonalisation. C'est le cas des machines universelles, ou subuniverselles. Schématiquement (MU désigne une machine universelle) :



De nombreuses difficultés concernant la notion identité personnelle (en particulier concernant son adéquation avec le mécanisme) disparaissent en définissant l'identité personnelle, non pas par une identité de style Leibnitzien, mais par un procédé d'évocation ou de construction de soi-même relatif à un niveau (sub) universel.

La différence essentielle est que l'identité Leibnitzienne est ontiquement nécessaire, vrai dans tous les mondes possibles, alors que l'identité "constructive" est relative à l'environnement universel.

Même si on peut prouver qu'une machine est solution de l'équation $\phi_x() = x$, il faut encore parier sur la stabilité de la machine universelle sous-jacente. Avec MDI, dans le cas de l'autoduplication, il n'est pas encore clair que l'on puisse naturellement s'identifier avec une telle solution, car il faudrait pour ça parvenir à identifier le niveau universel qui supporte la solution.

Ayant éventuellement choisi un niveau de duplication à partir de considérations empiriques, un mécaniste *pratiquant* sait, parce qu'il peut s'imaginer se rencontrer lui-même, qu'il sera difficile de prouver -de convaincre *l'autre*-, avoir survécu à une telle duplication (revoir 1.3).

L'existence d'un niveau, indépendamment de la problématique de son choix, permet cependant de considérer une identification de base entre la machine digitale (finiment descriptible, ayant une forme relative) et les systèmes formels.

2°) L'identification de base entre machine et système formel

Le philosophe Lucas est connu pour avoir tenté de réfuter l'hypothèse mécaniste au moyen des incomplétudes Gödéliennes. Il n'est pas le premier ; j'y reviens en détail au prochain chapitre. L'argument de Lucas repose sur une identification, que j'appelle l'identification de base, entre machine et système formel :

> *Gödel's theorem must apply to cybernetical machines, because it is of the essence of being a machine, that it should be a concrete instantiation of a formal system (Lucas 1961).*

De même MEC-IND impose l'identification de base au niveau où le fonctionnalisme est correct (c-à-d au niveau où les parties peuvent être fonctionnellement substituées. Dans ce cas, pour un être humain qui accepte les vérités classiques concernant les nombres naturels (et les entités codables par les nombres naturels), et qui croit ainsi à l'adéquation d'une théorie aussi faible que Q, les phénomènes Gödeliens sont en principe garantis et applicables.

A première vue, on ne voit pas ce que l'on pourrait tirer du couple $\phi_h(\ulcorner h \urcorner)$, tant la différence entre h et $\ulcorner h \urcorner$ semble gigantesque. Un million d'années ne suffirait pas à lecture de $\ulcorner h \urcorner$, par un humain h, et on imagine mal h s'identifier à $\ulcorner h \urcorner$.

D'autre part, dans l'expérience par la pensée de la reconstitution postposée, celui qui contemple son cristal de Gödel, contemple un codage de lui-même, au niveau (par définition) adéquat.

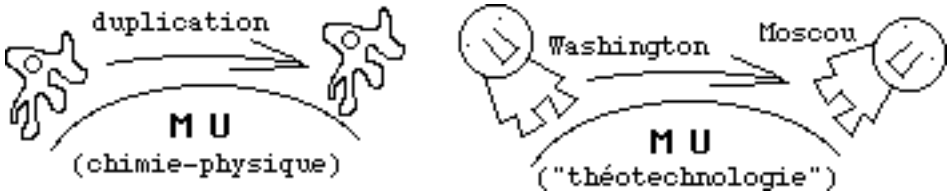
De même la collectivité humaine, par ses neurophysiologues interposés ou par ses généticiens, explorent à leur façon la représentation de l'homme relativement à son environnement.

Les résultats de l'informatique théorique peuvent contribuer à l'éclaircissement des difficultés conceptuelles des relations relatives entre h et ⌜h⌝.

Dans le cas de la duplication postposée, après le décodage du cristal, il semble que h ait vu son code ⌜h⌝ interprété par un reconstructeur et éxécuté par, disons, les lois de la physique. Il observe alors ce qu'on pourrait appeler un autre h, identique à lui-même. Toutefois il lui est toujours impossible de s'identifier à ce h, qu'il observe de l'extérieur. Il est plus opportun de décrire ce qu'il observe par la trace de ⌜h⌝, la trace étant relative à l'environnement universel, et ⌜h⌝ décrivant non plus le cristal de Gödel, mais son corps physique. Le caractère non nécessaire de la préservation de l'identité personnelle est capturée intuitivement par la confrontation entre h et la trace de ⌜h⌝. De l'extérieur, tout semble formel car on ne perçoit de façon directe que la forme des choses.

⌜h⌝ peut donc représenter aussi bien des descriptions de bas niveau, comme le codage sur le cristal de Gödel, que représenter, à un niveau plus élémentaire une représentation de soi plus fine. Dans aucun des cas ne peut-on se convaincre du bien-fondé d'une identification. Mais, par MEC-IND, au niveau de substitution adéquat, les références sont appropriées et l'incomplétude Gödélienne y est pertinente[34].

J'ai beaucoup insisté sur l'amibe et la duplication de soi :



que l'on peut capturer par une solution de l'équation :

$$\phi_x() = x \quad \text{(où } \{\phi_i\} \text{ est définie par MU[35])}.$$

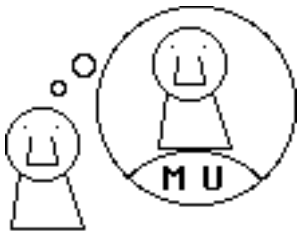
Et j'ai montré comment traiter des cas d'identités plus sophistiquées, comme celles qui sont distribuées dans un tissu (cf les planaires). Cette reproduction modélise aussi la duplication de soi de la section (1 3). Elle sont auto-appropriées par construction.

L'être humain se distingue de l'amibe en étant (au moins, même sans l'hypothèse mécaniste) une machine (potentiellement) universelle. Celle-ci lui permet de se représenter, ou de représenter une de ses traces :

---

[34] Remarquons qu'avec subst-sauf-quote, on est de la même façon passé des nombres naturels (traité par des programmes LISP, à des listes traitées par ce qui de l'extérieur sont des listes.

[35] Qui ne doit pas nécessairement être une numérotation acceptable, cf Rogers 1958.



Ce qui est capturable, de façon très grossière et non communicable (voir 2.3), par une récursion élémentaire, la trace opérant sur un niveau (relativement adéquat) :

$$\phi_e(x) = \phi_u(e,x), \text{ ou } \phi_e(x) = \text{trace '}((e))$$

L' hypothèse de la *représentation* présentée et discutée par quelques philosophes de l'esprit, comme Fodor 1987 ou Schiffer 1987, combinée au mécanisme indexical, permet donc de concevoir un soi représenté sous la forme d'une équation indexicale relative. Il permet d'étendre l'usage de la récursion de la biologie abstraite (Myhill 1964) à la psychologie abstraite[36](Myhill 1952) et est conforme à l'hypothèse forte de l'IA (STRONG AI thesis) :

FIA:

Esprit / cerveau $\approx$ programme / ordinateur

L'hypothèse de représentation est plus forte cependant que MEC-IND, mais avec MEC-**D**IG-IND, l'hypothèse de représentation est d'office satisfaite au niveau adéquat pour la substitution. On reste ouvert à l'idée que ce niveau soit très bas, et que pour survivre à une translation il faille, dans les "pires" des cas, dupliquer tout l'univers (physique) ou même une hiérachie transfinie de super-univers (comme des branches d'une interprétation du style d'Everett de la mécanique quantique) :

Esprit / univers $\approx$ programme / ordinateur

Le mécanisme digital est *représentationnel* sur le niveau fonctionnel adéquat.

Dans ce cas une duplication du style des expériences par la pensée décrites en (1 3) se laissent (grossièrement) capturer par

[36] Post 1921 aperçoit déjà une lois naturelle dans sa formulation anticipative de la thèse de Church.

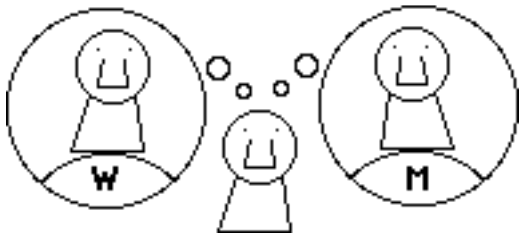
$$\phi_e(x) = (\text{si } x = \text{dup, (dovetelle ((e W)) ((e M))), sinon ...})$$

par exemple :

```
(def 'K-dove-F '(lambda (x data)
      (cond ((null data) nil)
            ((equal (car data) 'dup)
             (dovetelle (list (list x (cons 'W (cdr data))) (list x (cons 'M (cdr data)))))
            (t (x (cdr data))))))

(def 'k-dove (k dove-f))
```

Je vais, pour faciliter l'exposé de l'argumentation, souvent limiter la taille de la machine universelle capable de véhiculer l'identité personnelle à la taille du cerveau[37].

Je représenterai aussi l'activité d'une machine *(K dove)*, c'est-à-dire une machine qui se dovetelle elle-même, avec des notations BD :



L'opérateur de Kleene permet de supprimer les références personnelles dans le traitement de la duplication de soi. Il rester à analyser ce que le sujet peut prouver ou inférer sur les extensions qu'il imagine sur base de l'hypothèse mécaniste. (voir 2.3). Notons que les solutions indexicales sont automatiquement, *relativement au choix du niveau*, auto-appropriées. Si le sujet est sain et Σ_1-complet, les conséquences Gödéliennes doivent être prises en compte.

3°) La thèse de Church et le mécanisme

Brièvement l'argument de Lucas 1961 est le suivant[38]. Etant donné une machine, celle-ci étant équivalente à un système formel, je peux, en diagonalisant à la façon de Gödel générer et me convaincre (donc prouver dans le sens intuitif) une proposition vraie, que la machine ne peut pas prouver. Ce raisonnement marche quelle que soit la machine, donc l'ensemble des vérités prouvables par chaque machine est différent de l'ensemble des vérités que je sais prouver, donc je ne suis pas une machine.

---

[37] Accessoirement, dans 3.1, je donne des arguments empiriques en faveur de cette hypothèse. En 3.2 et 3.3, je donne des arguments qui suggèrent que cette hypothèse doit être considérablement nuancée.

[38] Ceci est repris en 2.3.1

L'argument de Lucas nécessite une préalable identification entre les systèmes formels et les machines auxquelles les diagonalisations de Gödel s'appliquent. Ceci montre aussi que la réfutation de Lucas repose sur la thèse de Church.

Dès lors, une façon de sauver le mécanisme contre l'argumentation Gödélienne de Lucas consiste à nier la thèse de Church. C'est plus ou moins le point de vue du mécanisme non digital de Searle 1980, 1984. Il identifie mal les niveaux et, à moins d'admettre des machines qui utilisent des infinis actuels ou plus généralement qui violent les principes de Gandy (voir 1.1), un mécanisme non digital *à-la-Searle* est a priori immune contre les réfutations basées sur l'usage du théorème de Gödel. Webb lui-même commencera aussi à développer cette idée (Webb 1968). L'analyse de Gandy et Sepherdson rend cette attitude très peu plausible[39]. Mais Webb 1980 réalisera après une réflexion approfondie sur la thèse de Church que celle-ci est plutôt un *ange gardien* du mécanisme digital. Cet aspect sera creusé dans la prochaine section.

L'idée principale est que pour rendre son argumentation vraiment convaincante, Lucas n'a pas d'autres choix que d'user du caractère effectif de sa réfutation, c'est-à-dire du caractère constructif de la démonstration du théorème de Gödel. Cette possibilité résulte de la mécanisabilité de l'argument de la diagonale. Avec la thèse de Church, cette effectivité est capturable par une machine de Turing (ou un programme LISP, etc.).

Avec cette effectivité, à partir d'une présentation (intensionnellement) convenable d'une théorie (ou d'une machine saine et consistante) énonçant des théorèmes, l'argument de Lucas est automatisable (voir section suivante), et même itérable dans le transfini constructif. Je montrerai qu'il est itérable au-delà du constructif dans un sens qui sera précisé.

J'appelle RL, pour *Réfutation de Lucas*, le programme qui génère, sur un argument x représentant une description d'une théorie T, la proposition indécidable, mais vraie, pour T. On est naturellement amener, comme Myhill 1964 et Case 1974, à profiter de 2-REC, pour écrire une machine "évoluant", démontrant une collection de plus en plus large de propositions vraies, en s'appliquant à elle-même la réfutation de Lucas.

---

[39] De même qu'elle rend suspecte l'invocation d'un parallélisme physique pour expliquer la conscience (Johnson Laird 1987). Que le parallélisme joue un rôle important en ce qui concerne la manifestation de la conscience pour une machine plongée dans un monde où de nombreux événements se passe en parallèle est plausible et compatible avec MDI. Mais MDI impose d'accepter l'émulation séquentielle de ce parallélisme comme suffisante pour que la conscience non seulement se manifeste, mais soit présente, pour autant que les entrées soient ralenties ou absentes, comme dans le rêve. C'est principalement ces considérations qui imposent la généralisation de Sepherdson des principes de Gandy (voir 1.1).

2.2.11 Machine de Myhill, machines variables et machines splittantes

La démonstration de l'incomplétude de M étant constructive, on peut augmenter M avec la capacité algorithmique de produire cet énoncé particulier. (avec RL). On obtient ainsi une nouvelle machine M_2 avec des capacités plus puissantes de prouvabilité. (comme avec Lucas ...) cette construction peut être itérée dans le transfini récursif (voir Turing 1939, Feferman 1962, 1988).

Avec l'extension de Case du théorème de Kleene il est possible de construire une suite croissante (par rapport à la capacité de prouvabilité) autoréférentielle de machines.

Supposons que x est le code de la machine M avec une partie explicitement reconnaissable comme étant la partie "prouveuse de théorème", alors M appartient au domaine des deux fonctions suivantes :

1) $\lambda x T(x)$, où T génère l'ensemble des théorèmes du système de preuve de x,

2) $\lambda x RL(x)$, où RL transforme x en une machine équivalente, excepté qu'il ajoute un énoncé (son code) vrai concernant x, mais non prouvable par x à l'ensemble des axiomes du système de preuves de x.

Nous voulons construire une suite de machines :

$$m_0, m_1, m_2, m_3...,$$

telle que chaque m_i, sur un argument particulier fixé "PROUVE", donne $T(m_i)$, et sur un autre "NEXT", donne m_{i+1}, où m_{i+1} est donné par $RL(m_i)$. m_0(PROUVE) genère les théorèmes d'une formalisation particulière en logique du premier ordre de l'arithmétique de Peano. Comme dans la planaire (infinie) le rôle de m_i est tenu par $\phi_e(i)$:

$\phi_{\phi_e(i)}(z)$ = si i=0 & z = PROUVE alors sortir T(PA),
sinon
si z=NEXT alors sortir $\phi_e(i+1)$, else
si z= PROUVE alors sortir $T(RL(\phi_e(i-1)))$.

e va de nouveau être obtenu en appliquant *k* sur une s-variante de la matrice de récursion:

t(x,i,z) = si i=0 & z = PROUVE alors sortir T(PA), else

si z=NEXT alors sortir $\phi_x(i+1)$, else
si z=PROUVE alors sortir $T(RL(\phi_x(i-1)))$. (3)

z représentant une liste quelconque, nous pouvons faire de chaque m_i une machine universelle :

si z=(UNIV x y) alors sortir $\phi_x(y)$.

Machine splittante : une machine peut produire une preuve non constructive concernant deux extensions d'elle-même. Par exemple une extension avec l'hypothèse que $\hat{}2^{\hat{}2}$ est rationnel, ou une extension avec l'hypothèse que $\hat{}2^{(\hat{}2^{\hat{}2})}$ est rationnel. Dans ce cas elle peut doveteller sur les deux hypothèses. En 2.3.5 on rencontrera des situations de ce genre, et je donnerai plus de renseignements.

2.2.12 Au sujet de quelques tentatives pour réfuter la thèse de Church, et l'apparition de propositions absolument indécidables

Réfuter la thèse de Church réfuterait, non pas le mécanisme flou de Searle, mais le mécanisme digitale.

Pour réfuter la thèse de Church, il suffit de trouver une fonction calculable, qui ne soit pas Turing, ou lambda, ou FORTRAN, ... calculable. En 1959, Kalmar[40] présente une fonction f, et argumente que : **soit** f est intuitivement calculable, et TC est réfutée, **soit** il existe une proposition arithmétique qui est *absolument indécidable.*

Il justifie que cette proposition absolument indécidable est vraie. Il en conclut que TC entraîne l'existence d'une proposition absolument (intuitivement) vraie et absolument (intuitivement) indécidable, *a very strange consequence indeed* (Kalmar 1959).

Je rappelle que :

$$K = \{x \mid x \in W_x\} = \{x \mid \phi_x(x)\Downarrow\} = \{x \mid \exists y\, T(x,x,y)\}$$

où T est le prédicat de Kleene. On sait que $x \notin W_x$ n'est pas RE, K^c,le complémentaire de K, n'est donc pas RE, et $\forall y \neg T(x,x,y)$ n'est pas RE.

Ainsi $\exists y\, T(x,x,y)$ est RE mais non récursif (cf le *théorème de Post*; un ensemble est récursif si et seulement si il est RE et son complémentaire est RE, voir 2.1).

En particulier la fonction totale :

---

[40] Voir aussi Kreisel 1970.

$$f(x) = \mu y\, T(x,x,y) \text{ si y existe}$$
$$= 0 \text{ si y n'existe pas.}$$

ne peut pas être calculée par une machine de Turing. Elle permettrait de décider $x \in W_x$. (On voit que $\mu y T(x,x,y)$ est un exemple de fonction partielle calculable non extensible en une fonction totale calculable).

Avec la thèse de Church f ne peut pas être intuitivement calculable. Pourtant, prétend Kalmar, voici un moyen, **intuitif**, de la calculer. Pour calculer la valeur de f en n, il suffit de doveteller sur

$$T(n,n,1),\ T(n,n,2),\ T(n,n,3),\ ....$$

et, *en même temps*, de chercher une preuve, pas nécessairement formelle (pourvu qu'elle soit correcte, convaincante et communicable) de

$$\neg \exists y T(n,n,y).$$

S'il existe un n tel que T(n,n,y), alors tôt ou tard le dovetellage des T(n,n,i) produira ce n. Si un tel n n'existe pas, alors tôt ou tard on trouvera une preuve intuitive de ce fait, et f(n) = 0. TC est ainsi réfutée.

A moins, concède Kalmar[41], qu'il n'existe pas de preuve intuitive de $\neg \exists y T(n,n,y)$. Cela montre que TC entraîne l'existence d'une proposition absolument indécidable. De plus, si une telle preuve intuitive n'existe pas, et que le dovetellage ne converge pas, $\neg \exists y T(n,n,y)$ est vraie, en plus d'être intuitivement (absolument) indécidable.

Cette conséquence est-elle aussi étrange que le prétend Kalmar ? Pour le mécaniste indexical, qui monte dans le translateur, une telle proposition a un air connu. Si MEC-IND est vrai, il n'est certainement pas intuitivement prouvable, puisque le dupliqué n'a aucun moyen intuitif de convaincre l'original de "sa" survie. Cela suggère, en particulier que le niveau *n* du mécanisme où le fonctionnalisme est correcte, puisse exister sans qu'il soit possible de trouver une preuve de l'adéquation du niveau. Cette suggestion va être approfondie dans la section suivante.

Si la conclusion, telle que Kalmar l'énonce, semble étrange, c'est qu'il a montré, avec P(n) pour $\neg \exists y\, T(n,n,y)$:

TC => nous savons qu'il existe un n tel que P(n) est vraie et P(n) est absolument indécidable,

---

[41] Kalmar semble tenir pour évident que l'ensemble des preuves intuitives est RE. C'est la thèse de Post-Turing PT. Voir footnote suivante.

et qu'il s'exprime comme s'il avait montré :

TC => il existe un n tel que nous savons que P(n) est vraie et nous savons que P(n) est indécidable,

ce qui serait contradictoire. Avec ⊡ pour savoir, cela revient donc à confondre

$$\boxdot \exists n\, P(n) \text{ avec } \exists n\, \boxdot P(n)$$

P(n) peut être vraie et absolument indécidable à partir du moment où n est absolument inconnu. Ce que Kalmar a, de fait, rigoureusement montré, c'est que TC entraîne l'existence de nombres ayant des propriétés bien définies mais absolument indémontrables[42].

2.2.13 constructif/non constructif

Une proposition existentielle du genre ∃xP(x), où on a ⊡∃nP(n) et ¬∃n ⊡ P(n), est dite *non constructive*. Si de plus ⊡ ¬∃n ⊡ P(n), elle est dite effectivement non constructive. C'est le cas pour la proposition de Kalmar (avec TC).

Nous avons rencontré d'autres phénomènes de non-constructivité. Le premier, dont nous sommes partis est : il n'existe pas de procédures pour distinguer l'indice d'une fonction totale calculable de l'indice d'une fonction partielle calculable. Puis nous avons observé le problème de l'arrêt, le castor occupé, etc... Dans chacun de ces cas, il est possible de justifier l'existence de proposition absolument indécidable par le procédé d'énumération des preuves intuitives de Kalmar. Ce n'est pas toujours le cas.

Par exemple, nous ne pouvons pas élargir constructivement une classe de complexité arbitraire

$$\boxdot\exists j(C_i \subsetneq C_j) \text{ n'entraîne pas } \exists j \boxdot (C_i \subsetneq C_j)$$

il *peut* donc exister un j tel que

$$\boxdot\exists j(C_i \subsetneq C_j) \;\&\; -\exists j \boxdot (C_i \subsetneq C_j)$$

mais pas *nécessairement*. On pourrait reconstruire l'argument de Kalmar et mettre en évidence une proposition absolument indécidable si on

---

[42] A strictement parler Kalmar a montré TC => il existe des propositions absolument indécidables ***ou*** l'ensemble des preuves intuitives n'est pas RE. On a donc PT => l'existence des propositions absolument indécidables (voir 2.3).

parvenait à justifier l'existence intuitive d'une procédure d'élargissement des classes de complexité.

C'est la même chose pour les phénomènes d'accélération de Blum. Ces phénomènes sont typiques de ce que rencontrent les Pythagoriciens de l'école du dehors. On peut montrer de façon assez précise que ces phénomènes n'apparaissent pas pour les écoles du dedans. Par exemple, Hartmanis montre que si on définit les classes de complexité par un équivalent constructif du genre :

$$CC_i = \{f \mid \exists j \; \boxdot(f = \phi_j \; \& \; \forall^{\infty}x \; \boxdot \; \beta_j(x) =< \phi_i(x))\}$$

alors les sauts de Borodin disparaissent[43]. Ils n'existent que pour les classes de complexité C_i qui sont différentes des CC_i.

Ces phénomènes, sous leurs formes absolues ou relatives, vont encore souvent apparaître dans ce qui va suivre[44].

2.2.14 La consistance de la thèse de Church intuitioniste

Kleene est aussi l'auteur d'une interprétation computationnelle de la version intuitioniste du constructivisme, telle que Heyting (voir 1.2) l'a formalisée (Kleene 1945, voir Kleene 1952). p_1 et p_2 sont des projection : $p_1(x,y) = x$, $p_2(x,y) = y$.

x **r** p	<=>	p, (p atomique)
x **r** p&q	<=>	$p_1(x)$ **r** p et $p_2(x)$ **r** p,
x **r** pvq	<=>	$p_1(x) = 0$ -> ($p_2(x)$ **r** p) & ($p_1(x) \neq 0$->$p_2(x)$ **r** q)),
x **r** p->q	<=>	$\forall y$(y **r** p -> ($\phi_x(y)\Downarrow$ & $\phi_x(y)$ **r** q),
x **r** $\exists$yA(y)	<=>	$p_2(x)$ **r** A($p_1(x)$),
x **r** $\forall$yA(y)	<=>	$\forall y(\phi_x(y)\Downarrow$ & $\phi_x(y)$ **r** A(y)),

où **r** est lu "réalise", x désigne un nombre naturel, codant éventuellemnt une paire de nombres naturels $<p_1(x), p_2(x)>$, ou une description d'une lambda-expression (machine de Turing, etc.).

---

[43] Un phénomène similaire apparaît avec certaines situations paradoxales en théorie des modèles. McCarty & Tennant montrent la façon dont le paradoxe de Skolem s'évanouit en mathématique constructive, j'y reviens dans 2.3.6.

[44] Finsler 1926 est connu pour avoir montrer informellement, et par diagonalisation l'existence de propositions indécidables. L'interprétation présentée ici du travail de Kalmar peut être vue comme une réhabilitation de Finsler. Mais Finsler n'est pas réellement réhabilitable car il fit à Gödel un procès de priorité, négligeant le fait que la preuve de Gödel s'applique exclusivement aux systèmes formels. Il prit donc le travail de Gödel comme une version *seulement plus rigoureuse* de son travail. Notre interprétation de Kalmar montre seulement que la thèse de Church permet une démonstration rigoureuse du travail de Finsler et l'existence d'une proposition absolument indémontrable. Le fait que Gödel, comme Finsler lui-même, négligea l'originalité de la preuve informelle de Finsler portant sur le(s) système(s) non formel(s) peut être attribué à la méfiance de Gödel vis-à-vis du mécanisme, comme je l'expose en 2.3.

On voit que le "ou" est une sorte de if-then-else. De même un nombre x réalise une implication p->q si et seulement si il existe une machine définie sur l'ensemble des réalisateurs de *p* à valeurs dans l'ensemble des réalisateurs de *q*. Un tel réalisateur transforme, en quelque sorte, une preuve de *p* en une preuve de q. Lorsqu'un nombre x réalise une proposition existentielle $\exists y A(y)$, ce nombre x code à la fois pour une valeur précise $p_1(x) = n$ vérifiant $A(\underline{n})$, et pour un réalisateur $p_2(x)$ de $A(\underline{n})$

Une proposition est dite réalisable s'il existe un nombre qui la réalise. Il n'est pas trop difficile, bien que long, de se convaincre que les tautologies propositionnelles intuitionistes sont réalisables (voir Kleene 1952).

Par exemple,

'(lambda (x) x) réalise p->p
'(lambda (x) (car x)) réalise p&q -> p
'(lambda (x) (car (cdr x))) réalise p&q -> q, etc.

en se souvenant que je travaille avec des *listes de Gödel* plutôt que des *nombres de Gödel*.

La formule ¬p est une abréviation de p->⊥ Un nombre qui réalise ⊥ est appelé une absurdité. On suppose qu'aucun nombre ne réalise ⊥, c-à-d qu'il n'y a pas d'absurdité. Il est facile de construire un réalisateur de p->((p->q)->q) : il faut construire une fonction partielle calculable envoyant un réalisateur de p sur un réalisateur de (p->q) -> q. Ce dernier est lui-même un nombre envoyant un réalisateur de p->q sur un réalisateur de q, c'est un applicateur, si bien qu'on trouve

(lambda (y) (list 'lambda (list (list x)) (list x y))),

Cette expression, appliquée à un réalisateur i_p de p, produit l'expression '(lambda (x) (x i_p)) qui, appliquée à un réalisateur $i_{p->q}$ de p->q, fournit un réalisateur de q.

En particulier, avec q = ⊥, on obtient un réalisateur de (p->((p->⊥)->⊥)), c'est-à-dire de (p-> ¬¬p). Par contre on ne trouvera pas de réalisateurs de (¬¬p->p). Un tel réalisateur transformerait un réalisateur de ¬¬p en un réalisateur de p. Mais un réalisateur de ¬¬p transforme lui-même un réalisateur de -p en un réalisateur de ⊥. Comme il n'y a pas de réalisateur de ⊥, il n'y a pas de réalisateur de ¬p. Nous pouvons seulement conclure qu'il n' y a pas de réalisateurs transformant un réalisateur de p en un réalisateur de ⊥. Nous savons que ¬p ne sera jamais réalisé, mais cela ne nous donne pas un réalisateur de p pour autant.

On peut construire un réalisateur du schéma d'induction en utilisant l'opérateur de Kleene.

La formule p(x) de Kalmar avec $p(x) = \exists y T(x,x,y)$ illustre, comme $p \rightarrow \neg\neg p$, une formule classiquement correcte, mais non constructive. En effet, informellement,la réalisabilité de $p(x) \vee \neg p(x)$ entraînerait la récursivité de $\exists y T(x,x,y)$.et ainsi, par exemple, la solubilité du problème de l'arrêt.

Plus curieux, la formule $\neg \forall x (p(x) \vee \neg p(x))$, qui est fausse classiquement, est réalisable. La raison est qu'un réalisateur de $\forall x (p(x) \vee \neg p(x))$ peut être transformé (récursivement) en un algorithme arrêt capable de décider le problème de l'arrêt, et qu'à partir de cet algorithme de l'arrêt, on peut expliciter une machine bien définie sur laquelle cet algorithme est incorrect, c'est-à-dire on peut mettre en évidence, de façon effective une absurdité (avec par exemple (k arrêt-moi-F) avec :

```
(def 'arrêt-moi-F '(lambda (x) (cond ((arrêt x) (infini))
                                     (t 'stop)) ))
```

Les propositions essentiellement non constructives seront d'une façon générale souvent représentables par des formules fausses classiquement, mais réalisables. Ce phénomène traduit ultimement la mécanisabilité des arguments diagonaux, et donc la faiblesse des arguments Gödeliens contre le mécanisme.

Last but not the least : la thèse de Church intuitioniste TCI est réalisable.

$$\forall x \exists y\, P(x,y) \rightarrow \exists z \forall x\, (\phi_z(x)\Downarrow \,\&\, P(x,\phi_z(x))) \qquad \text{TCI}$$

En effet, un réalisateur r_a du membre de gauche de TCI vérifie, en appliquant la définition de la réalisabilité de Kleene:

$$\forall x(\phi_{r_a}(x)\Downarrow \,\&\, p_2(\phi_{r_a}(x)) \;\mathbf{r}\; P(x, p_1(\phi_{r_a}(x))))$$

de même un réalisateur r_b du membre de droite doit vérifier (en se rappelant que $\phi_z(x)\Downarrow$ vers U(j) si et seulement si $\exists j T(z,x,j)$).

$$\begin{aligned}\forall x \;\; (\;\; & \phi_{p_2(r_b)}(x)\Downarrow \\ & \&\; p_1(p_2(\phi_{p_2(r_b)}(x)))) \;\mathbf{r}\; T(p1(r_b), x, p_1(\phi_{p_2(r_b)}(x))) \\ & \&\; p_2(p_2(\phi_{p_2(r_b)}(x))) \;\mathbf{r}\; P(x, U(p_1(\phi_{p_2(r_b)}(x)))) \quad)\end{aligned}$$

Il faut trouver un moyen de transformer un réalisateur r_a en un réalisateur r_b.

On peut se convaincre (avec une feuille et un crayon) qu'un réalisateur r_b est produit par

$$<\text{lambda}(x)\ p_1(\phi_{r_a}(x)),$$
$$\text{lambda}\ (x) <\mu j T(\text{lambda}(x)\ p_1\phi_{r_a}(x), x, j),$$
$$<0, p_2(p_1(\phi_{r_a}(x)))>>>$$

Cela permet de prouver la consistance de TCI dans les systèmes formels intuitionistes. En fait je démontrerai la consistance de la thèse de Post-Turing d'une tout autre façon. Si j'insiste sur la réalisabilité de Kleene, c'est plus pour la connexion qu'elle établit entre la théorie de la calculabilité et l'intuitionisme. Comme l'intuitionisme a déjà été relié à l'épistémisme (notamment par Gödel avec S4), cela relie l'épistémisme et la calculabilité.

Je mentionne encore les travaux de Kreisel et Kripke qui, en partant de la notion de *sujet créateur* introduite par Brouwer dans les fondements de l'analyse ont montré la consistance de la négation de la thèse de Church.

2.2.15 Conclusion

Bien qu'a priori la thèse de Church, aussi bien dans sa version intuitioniste et formalisable que classique et non formalisable puisse être considérée comme réductionniste, les résultats de limitations de Gödel et de Kalmar tempèrent substantiellement cette vision des choses. Avec l'identification de base cette tempérence est reportée sur l'hypothèse mécaniste.

La thèse de Church, que Post considérait comme une loi naturelle, fait des résultats d'incomplétudes, de véritables résultats rigoureux de la biologie et de la psychologie théorique. Myhill 1952 rappelle la portée *psychologique* de la convergence des travaux de Gödel, Church et Turing :

> *Not the least surprising aspect of their confirmation is the fact that three mathematicians working independently on the clarification of the concept of recognizability emerged with three seemingly totally unrelated analyses; the fact that these three were later discovered to be coextensive is, in itself, considering the diversity of their approaches, confirmation of an order rarely if ever encountered in psychology* (Myhill 1952).

Il aurait pu inclure Babbage pour son système de notation fonctionnelle, Post pour ses systèmes de production, Von Neuman pour l'ordinateur, Curry pour les combinateurs, Conway pour le jeu de la vie ou pour *Fractran*, Moore pour les systèmes physiques à n corps etc.

En fait le mot *machine* n'a plus le même sens depuis Gödel 1931 (et Church 1936, Turing 1936, Post 1921, 1944, etc.). Avant, on pensait qu'il était possible de construire une machine capable de prouver toutes les vérités concernant aussi bien les nombres naturels que les machines. A présent

nous savons que l'ensemble de ces vérités n'est pas axiomatisable, n'est pas formalisable, n'est pas récursivement énumérable.

Remarquons que le théorème de Gödel donne aussi l'opportunité de nous comparer aux machines, *sans user de l'identification de base*. Il suffit de prendre une théorie Σ_1-complète, et de comparer l'ensemble des théorèmes que *je* peux prouver intuitivement avec l'ensemble des théorèmes prouvable par une machine. La thèse de Post-Turing affirme l'égalité de ces deux ensembles. Cela permet une forme de test de Turing (Turing 1950) limité à un défi arithmétique. C'est ce défi que Lucas prétend être à même de toujours gagner contre la machine. Cela réfuterait MEC-DIG-BEH, et donc MEC-DIG-FORT, et donc MEC-DIG-IND.

Cependant la combinaison de l'incomplétude et de la thèse de Church rend l'hypothèse mécaniste non triviale. Comme ces remarques sont importantes au sujet de MDI, la section suivante commence par approfondir la relation entre les incomplétudes de Gödel 1931 et le mécanisme.

2.2.16 biblio-locale

BLUM M. and **MARQUES** I., **1973**, *On Complexity properties of Recursively Enumerable Sets*, Journal of Symbolic Logic, Vol 38, N° 4, pp. 579-593.

BOOLOS G., **1979**, *The Unprovability of Consistency, an Essay in Modal Logic*, Cambridge University Press.

BUCHSBAUM R., **1938**, *Animals Without Backbones*, second ed., Penguin Books Ltd, Harmondsworth, Middlesex, England, 1948, (first published in USA, 1938).

BUCHSBAUM R., **BUCHSBAUM** M.,**PEARSE** J., **PEARSE** V., **1987**, *Animals Without Backbones*, third ed., The University of Chicago Press, Chicago & London.

CALLUDE C., **1988**, *Theories of Computational Complexity*, North-Holland, Amsterdam.

CARROLL L., **1888**, *Curiosa Mathematica*, part 1, pp. ix-x, Londres. (Le passage est aussi cité dans Gattegno 1974)

CARSE J. P., **1986**, *Finite and Infinite Games*, Macmillan Publishing Company, New York, Traduction française, Seuil 1988.

CASE J., **1971**, *A Note on Degrees of Self-Describing Turing Machines*, Journal of the A.C.M., vol. 18, n° 3, pp 329-338.

CASE J., **1974**, *Periodicity in Generations of Automata*, in Mathematical Systems Theory. Vol. 8, n° 1. Springer Verlag, NY.

CHAITIN G. J., **1987**, *Information Randomness & Incompleteness*, 2ed Ed. 1990, World Scientific, Singapore.

CHURCH A.,**1936**, *An Unsolvable Problem of Elementary Number Theory*, The American Journal of Mathematics, 58, pp. 345-363. Repris dans Davis 1965, pp. 89-107.

COSSA P., **1955**, *La cybernétique "du cerveau humain aux cerveaux artificiels"*, Masson & Cie, Paris.

CUTLAND N. J., **1980**, *Computability An introduction to recursive function theory*, Cambridge University Press.

DAUBEN J. W., **1979**, *Georg Cantor, His Mathematics and Philosophy of the Infinite*, Princeton University Press, Princeton, New Jersey.

DAUBEN, J. W., **1977**, *Georg Cantor and Pope Leo XIII : mathematics, theology, and the infinite*, Journal of the History of Ideas, 38, pp. 85-108.

DAVIS, M., **1956**, *A note on universal Turing machines*, Automata Studies, Annals of mathematics studies, no 34, pp. 167-175, Princeton, N.Y.

DAVIS, M., **1957**, *The definition of universal Turing machines*, Proceedings of the American Mathematical Society, Vol 8, pp. 1125-1126.

DAVIS M. (ed.), **1965**, *The Undecidable*. Raven Press, Hewlett, New York.

DAVIS M., **1982**, *Why Gödel Didn't Have Church's Thesis*, Information and Control 54,.pp. 3-24.

FEFERMAN S., **1962**, *Transfinite Recursive Progressions of Axiomatic Theories*, Journal of Symbolic Logic, Vol 27, N° 3, pp. 259-316.

FEFERMAN S., **1988**, *Turing in the Land* of O(z), in Herken 1988.

FINSLER P., **1926**, *Formal Proofs and Undecidability*, in Van Heijenoort 1967, pp. 438-445.

FODOR J., **1987**, *Psychosemantics: The Problem of Meaning in the Philosophy of Mind*, A Bradford Book, The MIT Press, Cambridge.

GATTEGNO, J., **1974**, *Lewis Carroll une vie*, Seuil, Paris.

GÖDEL K., **1931**, *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I*, Monatsh., Math. Phys., 38, pp. 173-98, traduit en Français dans Le théorème de Gödel, Seuil, Paris, pp. 105-143, 1989, aussi en Anglais dans Davis 1965.

GÖDEL K., **1936**, *On the Length of Proofs*, traduit de l'allemand in Davies 1965, pp. 82-83.

GÖDEL K., **1946**, *Remarks before the Princeton Bicentennial Conference on Problems in Mathematics*, in DAVIS 1965, pp. 84-88.

HARTMANIS J. *Relations between Diagonalisation, Proof Systems, and Complexity Gaps*, Computer Science Department, Cornell University, Ithaca, New York 14853, USA.

HERKEN R. (ed), **1988**, *The Universal Turing Machine A Half-Century Survey*, Oxford University Press.

HOFSTADTER D., **1979**, *Gödel, Escher, Bach : an Eternal Golden Braid*, Basic Books, Inc., Publishers, New York.

JOHNSON-LAIRD P., **1987**, *How Could Consciousness Arise from the Computations of the Brain ?*, in Mindwaves, C. Blakemore and S. Greenfield, Basil Blackwell, Oxford and New York, pp. 247-257.

KALMAR L., **1959**, *An Argument against the Plausibility of Church's Thesis*, in Constructivity in Mathematics, Proceedings of the Amsterdam Colloquium 1957, A. Heyting (ed), North Holland, pp. 72-80.

KLEENE S. C., **1952**, *Introduction to Metamathematics*, North-Holland.

KREISEL, G., **1970**, *Church's thesis : a kind of reducibility axiom for constructive mathematics*, in Kino, A., Myhill, J., and Vesley, R.E. (eds.), Intuitionism and Proof Theory, Proceedings of the Summer Conference at Buffalo, New York, 1968, pp 121-150, North-Holland, Amsterdam.

KURTZ S. A., **MAHANEY** S. R., **ROYER** J. S., **1990**, *The Structure of Complete Degrees*, in Selman 1990.

LANGTON C. G., **1992**, *Life at the Edge of Chaos*, in Langton & Al. (Eds) 1992, pp. 41-91.

LANGTON C. G., **TAYLOR** C., **DOYNE FARMER** J., **RASMUSSEN** S., (Eds), 1992, *Artificial Life II*, A proceedings volume in the Santa Fe Institute Studies in the sciences of Complexity, vol X, Addison-Wesley.

LÖB M. H., **1955**, *Solution of a Problem of Leon Henkin*, Journal of Symbolic Logic, 20, pp. 115-118.

LUCAS J. R., **1961**, *Minds, Machines and Gödel*, Philosophy, vol. 36, pp. 112-127. (aussi dans Anderson)

McCULLOCH W. S., 1961, *What Is a Number, that a Man May Know It, and a Man, That He May Know a Number ?*, The Ninth Alfred Korzybski Memorial Lecture, General Semantics Bulletin, N° 26 & 27, Lakeville, Conn., pp. 7-18. Reprinted in McCulloch W. S., *Embodiments of Mind*, The MIT Press, 1988.

MONTAGUE R., **1962**, *Theories Incomparable with respect to Relative Interpretability*, The Journal of Symbolic Logic, Vol 27, N° 2, pp. 195-211.

MYHILL J., **1952**, *Some Philosophical Implications of Mathematical Logic*, The review of Metaphysics, Vol. VI, N° 2.

MYHILL, J., **1955**, *Creatives sets*, Zeitschrift für mathematische Logik und Grundlagen der Mathematik, vol 1, pp. 97-108.

MYHILL, J., **1964**, *Abstract theory of self-reproduction*. Views on General Systems theory, M.D. Mesarovic, Ed., Wiley, N.Y.pp. 106-118.

POST E., **1921**, *Absolutely Unsolvable Problems and Relatively Undecidable Propositions, Account of an Anticipation*, in Davis 1965, pp. 338-433.

POST E., **1944**, *Recursively Enumerable Sets of Positive Integers and their Decision Problems*, Bull. Am. Math. Soc. 50, pp. 284-316. also in Davis 1965, pp. 304-337.

POUR-HEL M. B., **1970**, *A Recursion-theoretic View of Axiomatizable Theories*, Dialectica, 24, N° 4, pp. 267-276.

WILKES M.W., **1977**, *Babbage as a Computer Pioneer*, Historia Mathematica, 4, pp. 415-440.

ROGERS H., **1958**, *Gödel Numbering of the Partial Recursive Functions*, Journal of Symbolic Logic, 23, pp. 331-341.

ROGERS H.,**1967**, *Theory of Recursive Functions and Effective Computability*, McGraw-Hill, 1967. (2ed, MIT Press, Cambridge, Massachusetts 1987).

ROYER J. S., **1987**, *A Connotational Theory of Program Structure*, Lecture Notes in Computer Science n° 273, Springer-Verlag, Berlin.

ROSSER J. B., **1936**, *Extensions of some theorems of Gödel and Church*, Journal of Symbolic Logic, 1, pp. 87-91, aussi dans Davis 1965, pp. 230-235.

ROSSER J. B., **1955**, *Deux esquisses de logique*, Paris, Gauthier-Villars, Louvain, E. Nauwelaerts.

ROYER J. S., **1987**, *A Connotational Theory of Program Structure*, Lecture Notes in Computer Science n° 273, Springer-Verlag, Berlin.

ROYER J.S., **1989**, *Two Recursion Theoretic Characterizations of Proof Speed-ups*, The Journal of Symbolic Logic, 54, N° 2.

RUSSELL B., **WHITEHEAD** A. N., **1910**, Principia Mathematica, Paperback Edition to *56, Cambridge University Press, 1970.

RUYER R., **1966**, *Paradoxes de la conscience et limites de l'automatisme*, Editions Albin Michel, Paris.

SALOMAA A.,**1985**, *Computation and Automata*, Cambridge University Press, Cambridge.

SCHIFFER S., **1987**, *Remnants of Meaning*, A Bradford Book, The MIT Press, Cambridge.

SEARLE J.R., **1980**, *Minds, Brains and Programs* dans The Behavioral and Brain Sciences, vol 3, Cambridge University Press. (repris aussi dans Dennett & Hofstadter 1981).

SEARLE J.R.,**1984**, *Minds, Brains and Sciences*. British Broadcasting Corporation. Trad. Franç. chez Hermann, 1985.

SELMAN A. L. (Ed.), **1990**, *Complexity Theory Retrospective*, Springer-Verlag, New York.

SMITH A. R., **1971**, *Simple Computation-Universal Cellular Spaces*, Journal of the ACM, 18, pp. 339-353.

SMITH A. R., **1992**, *Simple Nontrivial Self-Reproducing Machines*, , in Langton & Al. (Eds) pp. 709-725.

SMORYNSKI, C., **1981**, *Fifty Years of Self-Reference in Arithmetic*, Notre Dame Journal of Formal Logic, Vol. 22, n° 4, pp. 357-374.

SMORYNSKI C., **1985**, *Self-Reference and Modal Logic.*, Springer Verlag.

SMULLYAN R., **1961**, *Theory of Formal Systems*, Princeton University Press.

SMULLYAN R., **1985**, *Modality and Self-Reference*, in Shapiro 1985, pp. 191-209.

SOARE, R. I., **1980**, *Recursively Enumerable Sets and Degrees*, Springer-Verlag, Berlin.

SOLOVAY R, **1985**, *Explicit Henkin Sentences*, Journal of Symbolic Logic, Vol 50, N° 1, pp. 91-93.

TROUILLARD J., **1972**, *L'un et l'âme selon Proclos*, société d'édition "les belles lettres", Paris.

TURING A., **1936**, *On Computable Numbers with an Application to the Entscheidungsproblem*, Proc. London Math. Soc. 42, pp. 230-265. Aussi dans Davis 1965.

TURING A., **1939**, *Systems of Logic based on Ordinals*, Proc. London Math. Soc. 45, pp. 161-228. Aussi dans Davis 1965, pp. 155-222.

TURING A., **1950**, *Computing Machinery and Intelligence*, Mind, Vol. LIX, N° 236. Aussi dans Anderson 1964 ou Dennett D.C. and Hofstadter D., 1981 (biblio générale).

VAN HEIJENOORT J., **1967**, *From Frege to Gödel*, Harvard University Press, Cambridge, England.

WATSON J.D.,**HOPKINS** N.H.,**ROBERTS** J.W., **STEITZ** J. A. and **WEINER** A.L. *Molecular Biology of the Gene*, fourth edition, (**1965**, 1970, 1976, 1987), The Benjamin/Cummings Publishing Company, Inc. Texte Français élaboré par H. Costinesco, InterEditions, Paris, 1989.

WEBB J. C., **1968**, *Metamathematics and the Philosophy of Mind*, philosophy of sciences, XXXV, pp. 156-178.

WEBB J. C., **1980**, *Mechanism, Mentalism and Metamathematics : An essay on Finitism*, D. Reidel, Dordrecht, Holland.