Théorie des fonctions récursives

François Schwarzentruber

ENS Rennes

Table des matières

1 Fonctions primitives récursives

1.1 Syntaxe

On définit la **syntaxe** d'un langage $\mathcal{L}_{FPR}$ de **programmation fonctionnelle**.

Définition 1 (Langages des expressions des fonctions primitives récursives)
Le langage $\mathcal{L}_{FPR}$ des expressions des fonctions primitives récursives est défini par induction :
- $\mathbb{O}$ est une expression d'arité 0 ;
- σ est une expression d'arité 1 ;
- π_i^n où $n \in \mathbb{N}^*$ et $i \in \{1, \dots, n\}$ est d'arité n ;
- Si F est une expression d'arité n et $G_1, \dots, G_n$ sont des expressions d'arité ℓ alors : $comp(F, G_1, \dots, G_n)$ est une expression d'arité ℓ ;
- Si F est une expression d'arité n et G est une expression d'arité $n+2$ alors $rec(F, G)$ est une expression d'arité $n+1$.

1.2 Sémantique

Le programme $comp(\sigma, \pi_3^4)$ renvoie x_3 si on lui donne (x_1, x_2, x_3, x_4) en entrée. La fonction $\begin{array}{ccc} \mathbb{N}^4 & \to & \mathbb{N} \\ (x_1, x_2, x_3, x_4) & \mapsto & x_3 \end{array}$ est le **sens** du programme $comp(\sigma, \pi_3^4)$.

Définition 2 ()
On définit la fonction $\mathsf{sem} : \mathcal{L}_{FPR} \to \bigcup_{k \in \mathbb{N}} \mathcal{F}(\mathbb{N}^k, \mathbb{N})$ par induction structurelle :
- $\mathsf{sem}(\mathbb{O}) = \begin{array}{ccc} \emptyset & \to & \mathbb{N} \\ / & \mapsto & 0 \end{array}$;
- $\mathsf{sem}(\sigma) = \begin{array}{ccc} \mathbb{N} & \to & \mathbb{N} \\ x & \mapsto & x+1 \end{array}$;
- $\mathsf{sem}(\pi_i^n) = \begin{array}{ccc} \mathbb{N}^n & \to & \mathbb{N} \\ (x_1, \dots, x_n) & \mapsto & x_i \end{array}$;
- Si F est une expression d'arité n et $G_1, \dots, G_n$ sont des expressions d'arité ℓ alors $\mathsf{sem}(comp(F, G_1, \dots, G_n)) = \begin{array}{ccc} \mathbb{N}^\ell & \to & \mathbb{N} \\ \vec{x} & \mapsto & \mathsf{sem}(F)(\mathsf{sem}(G_1)(\vec{x}), \dots, \mathsf{sem}(G_n)(\vec{x})) \end{array}$;
- Si F est une expression d'arité ℓ et G est une expression d'arité $\ell+2$ alors $\mathsf{sem}(rec(F, G)) = g$ où $g : \mathbb{N}^{\ell+1}$ est la fonction définie par récurrence par :

$$g(\vec{x}, k) = \begin{cases} \mathsf{sem}(F)(\vec{x}) & \text{si } k = 0 \\ \mathsf{sem}(G)(\vec{x}, k-1, g(\vec{x}, k-1)) & \text{si } k > 0. \end{cases}$$

Définition 3 ()
Une fonction $\mathbb{N}^k \to \mathbb{N}$ est récursive primitive si elle est dans $\mathsf{sem}(\mathcal{L}_{FPR})$. On note $FPR = \mathsf{sem}(\mathcal{L}_{FPR})$.

Proposition 1 *L'ensemble des fonctions récursives primitives est dénombrable.*

Démonstration.
Le langage $\mathcal{L}_{FPR}$ est dénombrable. Donc $\mathsf{sem}(\mathcal{L}_{FPR})$ est dénombrable. ■

Remarque 1 *On peut fabriquer la fonction nulle d'arité 1 comme suit :* $\mathbb{O}^1 = rec(\mathbb{O}, \pi_2^2)$. *Ainsi, on peut définir la fonction nulle d'arité quelconque. Par exemple, la fonction nulle d'arité 4 est définie par le programme* $\mathbb{O} = comp(\mathbb{O}^1, \pi_1^4)$ *: il renvoie* 0 *si on lui donne* (x_1, x_2, x_3, x_4) *en entrée. Par simplicité, on note* $\mathbb{O}$ *la fonction nulle quelque soit l'arité.*

1.3 Schémas primitifs codés en Javascript

On peut voir les expressions des fonctions primitives récursives comme un sous-ensemble de Javascript.

1.3.1 Fonction nulle

```
function nul(x)
{
    return 0;
}
```

1.3.2 Fonction successeur

```
function succ(x)
{
    return x+1;
}
```

1.3.3 Fonctions projections

```
function projin(x1, ..., xn)
{
    return xi;
}
```

1.3.4 Composition

```
function comp_f_g1_gk(x1, ... xn)
{
    return f(g1(x1, ..., xn), ...gk(x1, ..., xn));
}
```

en supposant que les fonctions `f`, `g1`, ...`gk` soient définies.

1.3.5 Récursivté

```
function rec_f_g(x1, ..., xn, k)
{
    var r = f(x1, ..., xn);

    for(var i = 1; i <= k; i++)
    {
        r = g(x1, ..., xn, i-1, r);
    }

    return r;
}
```

en supposant que les fonctions `f` et `g` soient définies.

2 Exemples de fonctions récursives primitives

Voir : `http://people.irisa.fr/Francois.Schwarzentruber/recursive_functions/`

2.1 Prédicats

On peut interpréter 0 comme faux et 1 comme vrai. On peut écrire des fonctions primitives récursives qui sont des **prédicats** (estpair, $\geq$, etc.) puis des fonctions primitives récursives pour les connecteurs booléens (et, ou non). De même, on peut écrire des conditionnelles $ifthenelse(cond, iftrue, iffalse)$.

2.2 Minimisation bornée

Si F expression d'arité $\mathbb{N}^{\ell+1}$, on cherche une expression dont la sémantique est

$$\begin{array}{lcl} \mathbb{N}^{\ell+1} & \to & \mathbb{N} \\ (\vec{x}, n) & \mapsto & \text{le plus petit } i \in \{0, \ldots, \underline{\underline{n}}\} \text{ tel que } \mathsf{sem}(F)(\vec{x}, i) \neq 0 \end{array}.$$

2.3 Récurrence multiple

[[Deh00], p. 181]

2.3.1 Problématique

Question : est-ce que la fonction fib définie par $fib(0) = fib(1) = 1$ et

$$fib(k+2) = fib(k+1) + fib(k)$$

est primitive récursive ?

> Reformulation : attrape-t-on plus de fonctions récursives primitives en ajoutant un schéma de récursion à plusieurs variables ?

On peut se ramener un schéma de récursion à plusieurs variables de la façon suivante. On pose $\vec{f}(k) = (fib(k), fib(k+1))$. On a alors :

— $\vec{f}(0) = (1, 1)$;
— $\vec{f}(k+1) = \vec{h}(k, \vec{f}(k))$ où $\vec{h}(k, \vec{x}) = (x_2, x_1 + x_2)$.

Plus généralement, si $\vec{g} : \mathbb{N}^{\ell} \to \mathbb{N}^{q}$ est récursive primitive (i.e. chaque g_i est récursive primitive) et $\vec{h} : \mathbb{N}^{\ell+q+1} \to \mathbb{N}^{q}$ est récursive primitive, et que l'on définit $\vec{f} : \mathbb{N}^{\ell+1} \to \mathbb{N}^{q}$ par :

— $\vec{f}(\vec{n}, 0) = \vec{g}(\vec{n})$;
— $\vec{f}(\vec{n}, k+1) = \vec{h}(\vec{n}, k, \vec{f}(\vec{n}, k))$

est-ce que les projections de $\vec{f}$ sont récursives primitives ?

2.3.2 Solution : une bijection entre $\mathbb{N}^q$ et $\mathbb{N}$

Ce que la définition nous permet (schéma de récursion à une variable) :	Ce que l'on veut représenter :
— $f(\vec{n}, 0) = g(\vec{n})$ — $f(\vec{n}, k+1) = h(\vec{n}, k, f(\vec{n}, k))$	— $\vec{f}(\vec{n}, 0) = \vec{g}(\vec{n})$; — $\vec{f}(\vec{n}, k+1) = \vec{h}(\vec{n}, k, \vec{f}(\vec{n}, k))$

Si on a une bijection φ de $\mathbb{N}^q$ dans $\mathbb{N}$ qui est primitive récursive, alors la fonction $c = \varphi(\vec{f})$ est aussi primitive récursive :

— $c(\vec{n}, 0) = \varphi(\vec{g}(\vec{n}))$;
— $c(\vec{n}, k+1) = H(\vec{n}, k, c(\vec{n}, k)))$
où $H(\vec{n}, k, r) = \varphi(\vec{h}(\vec{n}, k, \varphi^{-1}(r)))$.

2.3.3 Bijection est primitive récursive

Théorème 1 *Il existe une bijection de $\mathbb{N}^2$ dans $\mathbb{N}$ qui est primitive récursive et dont l'inverse est aussi primitive récursive.*

Démonstration.
Définition Soit $\varphi(x, y) = \underbrace{2^x \times (2y+1)}_{\psi(x,y)} - 1$.

φ est une bijection La fonction ψ est bijection de $\mathbb{N}^2$ sur $\mathbb{N}^*$ car tout nombre non nul se décompose de façon unique en le produit d'une puissance de deux et d'un nombre impair. Le '-1' est là pour réparer le "non nul" et φ est une bijection de $\mathbb{N}^2$ dans $\mathbb{N}$.

φ est récursive primitive La fonction φ est bien récursive primitive comme composition de fonctions qui le sont. Montrons que φ^{-1} est récursive primitive.

φ^{-1} est une bijection La réciproque est de ψ est $\psi^{-1}(n) = (X(n), Y(n))$ avec :
— $X(n) =$ l'exposant de 2 dans la décomposition en nombre premier de n ;
— $Y(n) =$ la partie entière du nombre impair dans la décomposition en nombre premier de n.

φ^{-1} est récursive primitive **Retrouver l'abcisse.** X est le nombre de pas de divisions par 2 à réaliser à partir de n jusqu'à obtenir un entier impair. Un pas de calcul, c'est à dire une division lorsque c'est pair est $D(n) = \frac{n}{2}$ si n est pair et $= n$ si n est impair.

k pas de calcul se faire en itérant : on calcule n, puis $D(n)$, puis $D^2(n)$, etc. puis on s'arrête lorsque $D^k(n) = D^{k+1}(n)$. Soit $I(n, k)$ défini par :
— $I(n, 0) = n$
— $I(n, k+1) = D(I(n, k))$.
On a :

$$X(n) = \sum_{i=0}^{n} \mathbf{1}_<(I(n, i+1), I(n, i)).$$

Retrouver l'ordonnée. Une chose sûre, quand on itère trop (n fois par exemple !) le calcul précédent, on stagne sur le nombre impair. Il n'y a plus qu'à le rediviser par deux encore une fois pour avoir 'y' :

$$Y(n) = \lfloor \frac{I(n, n)}{2} \rfloor.$$

■

On étend alors à une bijection de $\mathbb{N}^q$ dans $\mathbb{N}$:

$$\varphi_q(n_1, \ldots, n_q) = \varphi(n_1, \varphi_{q-1}(n_2, \ldots, n_q)).$$

2.4 Structure de données : exemple des listes

Dans cette section, on montre comment coder une liste d'entiers avec des entiers. Les opérations d'encodage et les opérations sont toutes primitives récursives.

2.4.1 Encodage

Voici comment on encode une liste par un entier :
— $c([]) = 0$
— $c(x :: L) = 1 + \varphi(x, c(L))$ où $\varphi : \mathbb{N}^2 \to \mathbb{N}$ est la bijection du théorème 1.

2.4.2 Opérations

On code les opérations sur les listes de la façon suivante où x est un entier et ℓ est un entier qui représente une liste :
— $nil() = 0$: retourne la liste vide ;
— $cons(x, \ell) = 1 + \varphi(x, \ell)$ où $\varphi : \mathbb{N}^2 \to \mathbb{N}$ est la bijection du théorème 1 ;
— $head(\ell) = X(\ell - 1)$;
— $tail(\ell) = Y(\ell - 1)$ où X et Y sont les fonctions projections de φ^{-1}.

Exercice 1 *Ecrire length, élément numéro n, appartenance, etc.*

3 Langage de programmation impératif jouet vers fonctions primitives récursives

Considérons un programme P qui possède des variables globales $x_1, \ldots, x_n$ qui stockent des entiers positifs. Sans perte de généralité, on peut supposer que l'entrée du programme est stockée dans x_1 et que $x_2 = x_3 = \cdots = x_n = 0$. Un programme est un bloc d'instructions et la valeur de retour est écrite dans x_1.

Les valeurs prises par $(x_1, \ldots, x_n)$ est l'environnement du programme. On le code par un entier e. Il existe une fonction primitive récursive d_i pour obtenir x_i à partir de e. Il existe une fonction primitive récursive c pour calculer e à partir des valeurs de $(x_1, \ldots, x_n)$.

3.1 Constructions des programmes

Les constructions sont :
— Affectations :

```
xi = F(x1, ..., xn)
```

où F est une expression d'une fonction primitive récursive.
— Conditionnelle :

```
if(xi != 0)
{
    BLOC
}
```

où `BLOC` est un bloc d'instructions.

— Boucle for :

```
repeter xi fois
{
    BLOC
}
```

où `BLOC` est un bloc d'instructions.

3.2 Traduction en expression d'une fonction récursive primitive

— Une affectation

```
xi = F(x1, ..., xn)
```

est traduite $comp(c, d_1, \ldots, d_{i-1}, comp(F, d_1, \ldots, d_n), d_{i+1}, \ldots, d_n)$;

— La conditionnelle

```
if(xi != 0)
{
    BLOC
}
```

est traduite $ifthenelse(comp(notzero, d_i), bloc, \pi_1^1)$ où $bloc$ est la traduction de `BLOC` ;

— La boucle for

```
repeter xi fois
{
    BLOC
}
```

est traduite $comp(rec(\pi_1^1, comp(bloc, \pi_3^3)), \pi_1^1, d_i)$ où $bloc$ est la traduction de `BLOC`.

— Une séquence de `BLOC1 ; BLOC2` est traduite par $comp(bloc2, bloc1)$ où $bloc1, bloc2$ sont respectivment la traduction de `BLOC1` et `BLOC2`.

Un programme P prend en entrée x_1 et retourne x_1. Il est traduit donc par

$$comp(d_1, comp(p, comp(c, \pi_1^1, \mathbb{O}, \ldots, \mathbb{O})))$$

où p est la traduction du programme P.

4 Limite des fonctions récursives primitives

Il existe des fonctions calculables (par une machine de Turing) mais non primitives récursives.

4.1 Preuve via un argument diagonal

Théorème 2 *Il existe des fonctions calculables (par une machine de Turing) mais non primitives récursives.*

Démonstration.

Considérons une **énumération effective** $(e_n)_{n\in\mathbb{N}}$ des expressions du langage $\mathcal{L}_{FPR}$ d'arité 1. On considère

$$g : \begin{array}{ccc} \mathbb{N} & \to & \mathbb{N} \\ n & \mapsto & \mathsf{sem}(e_n)(n)+1 \end{array}$$

.

La fonction g est calculable. Voici un algorithme qui calcule g :

calculer$g(n)$:

1. Enumérer des expressions $e_1, e_2, \ldots$ jusqu'à obtenir e_n ;
2. Interpréter l'expression e_n pour calculer $\mathsf{sem}(e_n)(n)$
3. Ajouter 1

La fonction g n'est pas primitive récursive. Par l'absurde. Supposons que g est récursive primitive. Il existe k tel que $g(x) = \mathsf{sem}(e_k)(x)$ pour tout x. En particulier, $g(k) = \mathsf{sem}(e_k)(k) = \mathsf{sem}(e_k)(k)+1$. Contradiction.

■

4.2 Vers une fonction trop rapide : Ackermann-Péter

Cherchons une fonction qui n'est pas récursive primitive, mais qui pourtant a l'air calculable, en construisant des fonctions dont la croissante est de plus en plus rapide.

$a+b = a+\underbrace{1+1+\cdots+1}_{b \text{ examplaires}}$

$a\times b = \underbrace{a+a+\cdots+a}_{b \text{ exemplaires}}$

$a^b = a\uparrow^1 b = \underbrace{a\times a\times\cdots\times a}_{b \text{ exemplaires}}$

$a\uparrow^2 b = \underbrace{a^{a^{\cdot^{\cdot^{\cdot^{a}}}}}}_{b \text{ exemplaires}}$

De manière générale :

$a\uparrow^n b = \underbrace{a\uparrow^{n-1}\ldots\uparrow^{n-1} a}_{b \text{ exemplaires}}$

que l'on peut réécrire avec un schéma primitif récursif pour $n>1$:

$$a\uparrow^n b = \begin{cases} 1 & \text{si } b=0 \\ a\uparrow^{n-1}(a\uparrow^n (b-1)) & \text{sinon} \end{cases}$$

Exemple 1 *Voici quelques exemples de calcul :*

$2\uparrow^n 1 = 2$, $2\uparrow^n 2 = 4$ *pour tout* n.

Proposition 2 *Pour tout n, la fonction* $\uparrow^n : \begin{array}{ccc} \mathbb{N}^2 & \to & \mathbb{N} \\ (a,b) & \mapsto & a\uparrow^n b \end{array}$ *est récursive primitive.*

Démonstration.
Par récurrence sur n. ■
Par contre, ce n'est pas aussi clair (même faux) pour la fonction :

$$\begin{array}{rccc} f: & \mathbb{N}^3 & \to & \mathbb{N} \\ & (a,b,n) & \mapsto & a \uparrow^n b \end{array}$$

Pour simplifier, supprimons a comme argument et construisons une fonction jolie, $A : \mathbb{N}^2 \to \mathbb{N}$ définie par inductivement [1] :

— $A(0,m) = m+1$
— $A(k+1,0) = A(k,1)$
— $A(k+1,m+1) = A(k,A(k+1,m))$

Cette fonction est une variante de la fonction f définie précédemment. On peut montrer par récurrence que

$$A(k,n) = 2 \uparrow^{k-2} (n+3) - 3.$$

Son expression est explicite est bizarre mais sa définition donnée plus haut est plus simple. Il s'agit de la fonction d'Ackermann-Péter.

C'est une fonction qui est intuitivement calculable et dont le calcul termine bien qu'elle n'est pas en fait pas une fonction primitive récursive. Intuitivement, la récursivité d'Ackermann-Péter ne peut pas se ramener à un schéma primitif de récursivité via une bijection de $\mathbb{N}^2$ dans $\mathbb{N}$.

Théorème 3 *$A \notin FPR$.*

Démonstration.
[[Deh00], p. 192] Résumé de la démonstration :

1. On montre que les fonctions primitives récursives sont assez lentes (par induction structurelle).
2. Par ailleurs, $n \mapsto A(n,n)$ n'est pas lente. Donc $A \notin FPR$.

Par '$f : \mathbb{N}^\ell \to \mathbb{N}$ est lente', on entend la propriété $P(f)$ suivante :

il existe un entier k tel que pour tout $(n_1, \ldots, n_\ell) \in \mathbb{N}^\ell$, on a

$$f(n_1, \ldots, n_\ell) \leq A(k, \sum_{i=1}^{\ell} n_i).$$

■

5 Fonctions μ-récursives partielles

L'idée est que les fonctions primitives récursives correspondent aux programmes que l'on peut écrire avec des boucles **for** et des **if** . Mais il manque les boucles **while** .

1. L'induction est ici défini sur l'ordre lexicographique sur $\mathbb{N}^2$

5.1 Syntaxe

On rajoute la minimisation non bornée de F, noté $mu(F)$.

Définition 4 (Langages des expressions des fonctions μ-récursives)
Le langage $\mathcal{L}_{F\mu R}$ des expressions des fonctions μ-récursives sont définies par induction :

- $\mathbb{O}$ est une expression d'arité 1 ;
- σ est une expression d'arité 1 ;
- π_i^n où $n \in \mathbb{N}^*$ et $i \in \{1, \dots, n\}$ est d'arité n ;
- Si F est une expression d'arité n et $G_1, \dots, G_n$ sont des expressions d'arité ℓ alors : $comp(F, G_1, \dots, G_n)$ est une expression d'arité ℓ ;
- Si F est une expression d'arité n et G est une expression d'arité $n+2$ alors $rec(F, G)$ est une expression d'arité $n+1$;
- Si F est d'arité $\ell + 1$ alors
  $mu(F)$ est une expression d'arité ℓ.

5.2 Sémantique

Définition 5 ()
On définit la fonction $\mathsf{sem} : \mathcal{L}_{FPR} \to \bigcup_{k\in\mathbb{N}} \mathcal{F}_{partielle}(\mathbb{N}^k, \mathbb{N})$ par induction structurelle :

- $\mathsf{sem}(\mathbb{O}) = \begin{array}{ccc} \mathbb{N} & \to & \mathbb{N} \\ x & \mapsto & 0 \end{array}$;
- $\mathsf{sem}(\sigma) = \begin{array}{ccc} \mathbb{N} & \to & \mathbb{N} \\ x & \mapsto & x+1 \end{array}$;
- $\mathsf{sem}(\pi_i^n) = \begin{array}{ccc} \mathbb{N}^n & \to & \mathbb{N} \\ (x_1, \dots, x_n) & \mapsto & x_i \end{array}$;
- Si F est une expression d'arité n et $G_1, \dots, G_n$ sont des expressions d'arité ℓ alors

$$\mathsf{sem}(comp(F, G_1, \dots, G_n)) = \begin{array}{ccc} \mathbb{N}^\ell & \to & \mathbb{N} \\ \vec{x} & \mapsto & \begin{cases} \mathsf{sem}(F)(\mathsf{sem}(G_1)(\vec{x}), \dots, \mathsf{sem}(G_n)(\vec{x})) \\ \quad \text{si } \mathsf{sem}(G_1)(\vec{x}), \dots, \mathsf{sem}(G_n)(\vec{x}) \text{ sont définis} \\ \text{non défini sinon} ; \end{cases} \end{array}$$

- Si F est une expression d'arité ℓ et G est une expression d'arité $\ell + 2$ alors $\mathsf{sem}(rec(F, G)) = g$ où $g : \mathbb{N}^{\ell+1}$ est la fonction définie par récurrence par :

$$g(\vec{x}, k) = \begin{cases} \mathsf{sem}(F)(\vec{x}) & \text{si } k = 0 \\ \mathsf{sem}(G)(\vec{x}, k-1, g(\vec{x}, k-1)) & \text{si } k > 0 \text{ et } g(\vec{x}, k-1) \text{ définie.} \\ \text{non défini} & \text{sinon} \end{cases}$$

- Si F est d'arité $\ell + 1$ alors

$$\mathsf{sem}(mu(F)) = \begin{array}{ccc} \mathbb{N}^\ell & \to & \mathbb{N} \\ \vec{x} & \mapsto & \begin{cases} \text{le plus petit } i \in \mathbb{N} \text{ tel que } \mathsf{sem}(F)(\vec{x}, i) \neq 0 \\ \qquad\qquad\qquad\qquad \text{si un tel } i \text{ existe} \\ \text{non défini} \qquad\qquad\qquad\quad \text{si un tel } i \text{ n'existe pas.} \end{cases} \end{array}$$

Remarque 2 *On note parfois*

$$\mu i.q(\vec{n}, i) = \begin{cases} \textit{le plus petit } i \in \mathbb{N} \textit{ tel que } q(\vec{n}, i) = 1 \\ \textit{non défini si un tel } i \textit{ n'existe pas} \end{cases}$$

Définition 6 ()
Une fonction partielle $\mathbb{N}^k \to \mathbb{N}$ est μ-récursive partielle si elle est dans $\mathsf{sem}(\mathcal{L}_{F\mu R})$.

5.3 Minimisation non bornée codée en Javascript

```
function muq(x1, ... xn)
{
    var i = 0;
    while(q(x1, ...xn) == 0)
    {
        i++;
    }
    return i;
}
```

en supposant que la fonction `q` est définie.

6 Langage de programmation impératif avec while vers fonctions μ-récursives partielles

6.1 Constructions des programmes

On ajoute aux constructions données en sous-section 3.1 la boucle while :

```
while(xi == 0)
{
    BLOC
}
```

6.2 Traduction en expression d'une fonction μ-récursive partielle

La construction

```
while(xi == 0)
{
    BLOC
}
```

est traduite par $comp(rep, \pi_1^1, mu(comp(d_i, rep)))$ où

— $rep = rec(\pi_1^1, comp(bloc, \pi_3^3))$ est interprété comme la fonction qui prend en argument l'environnement e et un nombre d'itérations i et qui retourne l'environnement après i itérations de `BLOC`.

7 Fonctions μ-récursives totales

C'est la minimisation non bornée qui est dangereuse : le calcul effectif de $\mathsf{sem}(mu(F))(\vec{x})$ peut ne pas terminer et donc la fonction n'est pas définie. Pour éviter cela, nous nous restreignons aux fonctions q dits *sûrs*.

Définition 7 (fonction sûre)
Une fonction $q : \mathbb{N}^{\ell+1} \to \mathbb{N}$ est sûre[2] si pour tout $\vec{n} \in \mathbb{N}^\ell$, il existe $i \in \mathbb{N}$ tel que $q(\vec{n}, i) \neq 0$.

2. [Wol06] introduit la notion de prédicat sûr.

Définition 8 ()
Une fonction μ-récursive totale est une fonction totale de la forme $\mathsf{sem}(e)$ où $e \in \mathcal{L}_{F\mu R}$.

Proposition 3 *Il existe des fonctions totales non μ-récursives totales.*

Démonstration.
L'ensemble des fonctions est non dénombrable alors que l'ensemble des fonctions μ-récursives totales est dénombrable. ■

Remarque 3 *Attention, cependant ! L'argument diagonal ne fonctionne pas. Existe-t-il une énumération* effective $(f_n)_{n\in\mathbb{N}}$ *des fonctions μ-récursives (totales) ?*

Proposition 4 *Il existe une fonction μ-récursive partielle f telle que pour aucune fonction récursive totale h, pour tout $\vec{n}$, on ait : $h(\vec{n}) = \begin{cases} f(\vec{n}) \text{ est défini} \\ 0 \text{ si } f(\vec{n}) \text{ non défini} \end{cases}$.*

Démonstration.
Plus tard... ■

8 Comparaison avec les machines de Turing

Définition 9 ()
Une machine de Turing calcule $f : \Sigma^* \to \Sigma^*$ si pour tout mot d'entrée x, la machine s'arrête dans une configuration où $f(x)$ se trouve sur le ruban.

8.1 Des fonctions μ-récursives aux machines de Turing

8.1.1 Codage des entiers dans une machine de Turing

On représente un entier par un mot sur l'alphabet $\{0, 1\}$ qui est représentation binaire. On représente un tuple sur l'alphabet $\{0, 1, , \}$: par exemple, $(2, 1, 0) \in \mathbb{N}^3$ est représenté par le mot $10, 1, 0$.

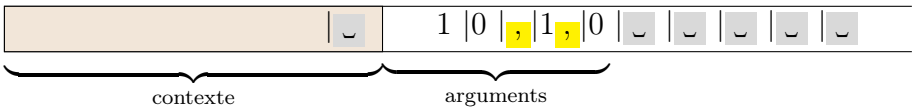
8.1.2 Théorème

Théorème 4 *Toute fonction récursive partielle f est calculable par une machine de Turing. Plus précisément, si on considère la configuration initiale avec l'entier w sur le ruban, si $f(w)$ est définie, la machine s'arrête et $f(w)$ est écrit sur le ruban à la fin et si $f(w)$ n'est pas définie, la machine ne termine pas.*

Corollaire 1 *Toute fonction récursive totale est calculable par une machine de Turing (qui s'arrête sur toutes les entrées).*

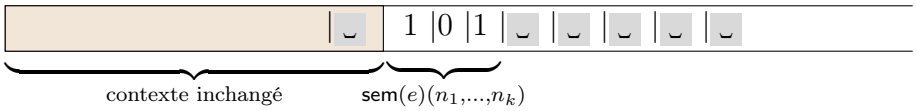
Démonstration.
On démontre le théorème 4. Pour tout expression $e \in \mathcal{L}_{F\mu R}$, on définit la propriété $\mathcal{P}(e)$ suivante :

Si e est d'arité k, alors il existe une machine de Turing M_e à un ruban tel que pour tout $(n_1, \ldots, n_k) \in \mathbb{N}^k$,
depuis toute configuration initiale avec sur le ruban



c'est à dire, il y a un contexte (le mot vide ou un mot sur $\{0, 1, |, \text{␣}, \text{,}\}$ avec jamais deux espaces ␣ consécutifs et qui finit avec un symbole ␣) puis à la fin, les nombres $n_1, \ldots, n_k$, écrits en binaire séparés par des ,

— la machine s'arrête avec le ruban suivant

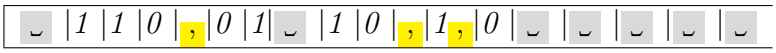


si $\mathsf{sem}(e)(n_1, \ldots, n_k)$ est défini ;

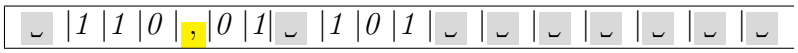
— la machine boucle si $\mathsf{sem}(e)(n_1, \ldots, n_k)$ n'est pas défini.

Le symbole ␣ est le symbole blanc aussi utilisé pour séparer les appels de fonctions, où 0 et 1 sont les chiffres utilisés pour coder les entiers en binaire, et , est utilisé pour séparer les arguments.

Exemple 2 *Si le ruban est comme suit :*



et que $f(2, 1, 0) = 5$ *alors après l'exécution de* M_f*, on a :*



Fonction nulle. Voici $M_{\mathbb{O}}$:

Remplace les arguments (il n'y a rien) par 0 tout à la fin

d'où $\mathcal{P}(\mathbb{O})$

Fonction successeur. Voici M_σ :

//il n'y a qu'un seul nombre après l'éventuel dernier ␣
Remplacer le dernier nombre écrit par lui-même plus 1

d'où $\mathcal{P}(\sigma)$

$i^{\text{ème}}$ projection à n arguments. Voici $M_{\pi_i^n}$:

Des n derniers nombres après le dernier ␣
Ajouter un ␣ et réécrire le $i^{\text{ème}}$ nombre après
Recopier le nombre à la fin au début de la zone des arguments
Effacer toute la suite

d'où $\mathcal{P}(\pi_i^n)$

Composition. Soit $g \in \mathcal{L}_{F\mu R}$ d'arité ℓ et $h_1, \ldots, h_\ell \in \mathcal{L}_{F\mu R}$ à d'arité k telles que $\mathcal{P}(g)$ et $\mathcal{P}(h_1), \ldots, \mathcal{P}(h_\ell)$ soient vraies. Montrons que $\mathcal{P}(comp(g, h_1, \ldots, h_\ell))$ est vraie. Voici la machine $M_{comp(g,h_1,\ldots,h_\ell)}$.

for $i := 1$ à ℓ **do**
  Recopier $\vec{n}$ (écrit $i-1$ portions de ruban avant) tout à la fin, précédé d'un $\sqcup$
  Appeler M_{h_i} (il remplace le $\vec{n}$ que l'on vient de recopier par $\mathsf{sem}(h_i)(\vec{n})$)
endFor
Décaler les résultats $\mathsf{sem}(h_1)(\vec{n}) \sqcup \ldots \sqcup \mathsf{sem}(h_\ell)(\vec{n})$ sur les $\vec{n}$ encore restants et efface la fin
Remplacer les $\ell - 1$ $\sqcup$ par des ,
Appeler M_g

Récursion. Soit $g \in \mathcal{L}_{F\mu R}$ d'arité ℓ et $h \in \mathcal{L}_{F\mu R}$ d'arité $\ell + 2$ telles que $\mathcal{P}(g)$ et $\mathcal{P}(h)$ soient vraies. Montrons que $\mathcal{P}(rec(g, h))$. Voici la machine $M_{rec(g,h)}$.

Ecrire $\sqcup$
Ecrire 0 (appelons cette portion i)
Ecrire $\sqcup$
Recopier k ici
Ecrire $\sqcup$
Recopier $\vec{n}$
Appeler M_g
while $i < k$ **do**
  Décaler $\mathsf{sem}(rec(g, h))(\vec{n}, i)$ pour insérer $\vec{n}$, i , avant
  Appeler M_h
  Incrémenter i de 1
endWhile
Décaler le résultat $\mathsf{sem}(rec(g, h))(\vec{n}, i)$ vers la gauche en effaçant $\vec{n}$, k, i

Minimisation non bornée Soit $q \in \mathcal{L}_{FPR}$ d'arité $\ell + 1$ tel que $\mathcal{P}(q)$. Montrons $\mathcal{P}(mu(q))$.

Ecrire $\sqcup$
Ecrire 0 (appelons cette portion i)
Ecrire $\sqcup$
Recopier $\vec{n}$, 0
Appeler M_q
while le résultat de M_q est 0 **do**
  Incrémenter i de 1
  Recopier $\vec{n}$, i à la place du précédent résultat de M_q
  Appeler M_q
endWhile
Supprimer le résultat de M_q
Décaler i vers la gauche (et supprimer $\vec{n}$)

■

Remarque 4 *Les minimisations non bornées avec prédicats non sûrs donnent des exécutions infinies.*

8.2 Des machines de Turing aux fonctions μ-récursives

8.2.1 Codage des entiers

— Mauvaise solution qui ne fonctionne pas. On pourrait représenter le ruban de la machine par le nombre correspondant à la représentation binaire écrite sur le ruban. Supposons que l'alphabet contient deux lettres $\{0, 1\}$, on aurait envie de le mot du ruban comme le nombre dont le mot est sa représentation binaire. Malheureusement, les mots 1, 01, 001 représentent tous l'entier 1. Alors que la machine peut avoir des exécutions différentes sur 1, 01, 001...

— Une possibilité est de considérer que l'alphabet est $\{1, 2\}$ puis de travailler en base 3 (c'est la solution du livre [Wol06][p. 173]).
C'est à dire une suite de 1 et 2 comme $w = w_0, \dots w_\ell$ encode le nombre :

$$gd(w) = \sum_{i=0}^{\ell} 3^i w_i.$$

— Soit on peut utiliser la structure de données Liste déjà vu pour encoder le ruban.

8.2.2 Théorème

Théorème 5 *Toute fonction calculable par machine de Turing avec éventuellement des exécutions infinies (où alors la fonction n'est pas définie) est μ-récursive partielle.*
Toute fonction calculable par machine de Turing est récursive totale.

Démonstration.
On peut écrire le comportement de la machine dans le langage donné à la section 6.1. Le programme a la forme suivante et on laisse le soin au lecteur de s'occuper des détails.

```
entree: variable ruban

etat = q0
curseur = 0

while(etat != qfinal)
{
        ruban, etat, curseur = nouvelleconfiguration(ruban, etat, curseur)
}
```

■

Démonstration.
Donnons une démonstration proche de celle donnée dans [Wol06] Considérons une machine de Turing. Les entiers représentent les mots et les configurations de la machine. On construit des fonctions primitives récursives suivantes :

— *init* : $\mathbb{N} \to \mathbb{N}$: celle qui associe la représentation d'un mot x à la représentation de la configuration initiale ;

— *ruban* : $\mathbb{N} \to \mathbb{N}$ une fonction qui à partir d'une configuration c renvoie le mot sur le ruban.

— *suivante* : $\mathbb{N} \to \mathbb{N}$: celle qui associe la représentation d'une configuration c à la représentation de la configuration suivante de c ;

— *suivante** : $\mathbb{N}^2 \to \mathbb{N}$ celle qui, plus générale, à une configuration c et un entier n associe la configuration que l'on a dans n étapes à partir de c ;

— $stop : \mathbb{N} \to \mathbb{N}$ un prédicat qui dit, étant donné une configuration c dit si elle est finale.

On utilise alors la minimisation non bornée pour définir la fonction suivante

$$nbEtapes : x \mapsto \mu i.stop(suivante^*(init(x), i))$$

qui associe au mot d'entrée le nombre d'étapes de calcul pour arriver dans un état final. C'est a priori une *fonction partielle*. Le prédicat $stop(suivante^*(init(x), i)$ n'est *pas forcément sûr* dans le cas général. En fait, il est sûr ssi la machine s'arrête quelque soit l'entrée.

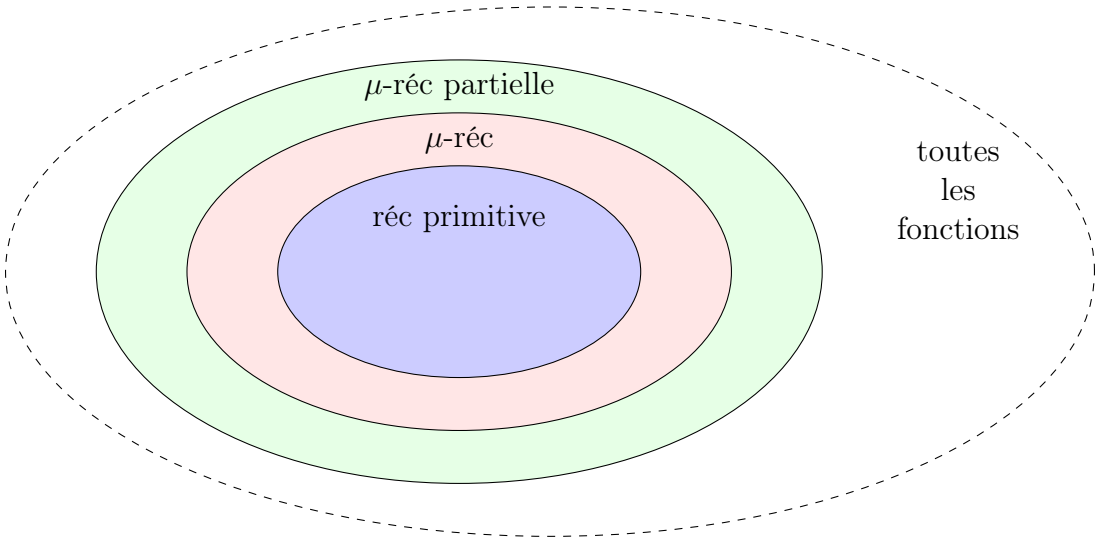
Voici alors la fonction qui calcule la même chose que la machine de Turing :

$$f : x \mapsto ruban(suivante^*(init(x), nbEtapes(x))).$$

■

Exercice 2 *Montrer qu'une fonction calculable par une machine de Turing en temps $O(T(n))$ où $T(n)$ est primitif récursif est aussi primitive récursive.*

9 Bilan



Fonctions récursives primitives	Fonctions μ-récursives totales	Fonctions μ-récursives partielles
on sait "syntaxiquement" que leurs calculs terminent toujours	on sait "sémantiquement" que leurs calculs terminent toujours	leurs calculs ne terminent pas forcément
programmes sans boucle *while*	programmes avec boucle *while*	
totales		partielles

10 Questions

Question 1 *Est-ce que la bijection "naturelle" $f : \mathbb{N}^2 \to \mathbb{N}$ est primitive récursive et d'inverse primitive récursive ?*

Question 2 *Existe-t-il une fonction $f : \mathbb{N} \to \mathbb{N}$ primitive récursive bijective telle que f^{-1} soit récursive mais pas primitive ?*

11 Notes bibliographiques

11.1 Syntaxe VS sémantique

Dans beaucoup de livres ([Wol06][p. 155], [Dehornoy, Mathématiques de l'informatique, p. 171], etc.), aucune syntaxe n'est généralement donnée pour les fonctions primitives récursives ou les fonctions μ-récursives partielles. Je n'y vois que des désavantages.

- Impossible de créer un outil.
- Des justifications basées sur la syntaxe qui sont vagues (dans [Wol06], p. 129, "chaque fonction primitive récursive peut être décrite par une chaîne de caractères".)
- Aucune vision synthétique de comment se compose une fonction primitive récursive une fonction la fonction construite.
- Difficulté d'écrire les traductions vues en sous-section 3.2 et 6.2.

Martin-Löf et Kleene semblent aussi introduire une syntaxe TODO : étude bibliographie à compléter.

11.2 Prédicats

Wolper [Wol06] introduit la notion de prédicat : c'est une fonction $\mathbb{N}^k \to \{0, 1\}$. Intuitivement, on interprète 0 comme faux et 1 comme vrai. Je n'aime pas car les définitions ne sont plus syntaxiques. La minimisation bornée et non bornée ne sont définies que pour les prédicats mais c'est une restriction sémantique.

11.3 Fonctions à valeurs dans $\mathbb{N}^k$

Ici, nos fonctions sont à valeurs dans $\mathbb{N}$ et au besoin nous codons un tuple de $\mathbb{N}^k$ avec la technique de la sous-section 2.3.2. Dans [Bro89], les fonctions sont à valeurs dans $\mathbb{N}^k$ et l'auteur introduit (p. 201) le schéma de **combinaison** : à partir de $f : \mathbb{N}^k \to \mathbb{N}^n$ et $g : \mathbb{N}^k \to \mathbb{N}^m$ on construit $\begin{array}{rccc} f \times g : & \mathbb{N}^k & \to & \mathbb{N}^{n+m} \\ & \vec{x} & \mapsto & (f(\vec{x}), g(\vec{x}) \end{array}$.

11.4 Fonctions partielles

Dans [Wol06], la définition de la minimisation non bornée ne donnent pas une fonctions partielles car lorsqu'un i n'existe pas, on définit $\mu i.q(\vec{n}, i) = 0$. La fonction est donc totale alors que le "calcul" ne devrait pas terminer. C'est confus.

11.5 Langages de programmation jouets

11.5.1 Sans while

Bucle. Dans [HHF85], il y a la présentation d'un langage appelé Bucle, sans boucle **while**. Le voici :

- Les variables sont entières positives ;
- des affectations $x =$ quelque chose de primitif récursif ;
- les variables déclarées non initialisées sont à 0 ;
- L'incrément ;
- Déclaration de fonctions ;
- Appel par valeur SANS récursion ;
- Boucles for du type : répéter x fois le bloc B (où x n'est pas modifié dans le bloc B)
- Le retour de fonction **return** .

Théorème 6 *Une fonction f est récursive primitive ssi il existe une fonction de Bucle qui calcule f.*

Modèle FOR. Un langage équivalent est montré dans [Calculateurs, calculs, calculabilité, Olivier Ridoux, Gives Lesventes, chap. 6] appelé le modèle FOR.

LOOP. Créé par Uwe Schöning TODO : mettre une source

11.5.2 Avec while

Mucle. Dans [HHF85], il y a la présentation d'un langage appelé Mucle, qui est une extension de Bucle avec une boucle MU.

Modèle WHILE. Un langage équivalent est montré dans [Calculateurs, calculs, calculabilité, Olivier Ridoux, Gives Lesventes, chap. 6] appelé le modèle WHILE.

Langage de programmation "Bare-bones". Langage de programmation "Bare-bones" [[Bro89] (p. 227)]

- incr x
- decr x
- boucle while $x \neq 0$
- entiers positifs
- clear x : mettre x à 0 (peut-être réalisé avec while $x \neq 0\{$ decr $x\}$)
- Pas de fonctions, pas d'appels.

11.6 Fonctions d'Ackermann

Dans la plupart des ouvrages, on parle de la fonction d'Ackermann. Il s'agit d'une simplification avec deux variables seulement dûe à Rózsa Péter.

11.7 Aller vraiment plus loin

Hiérarchie de Grzegorczyk.

Références

[Bro89] J Glenn Brookshear. *Theory of computation : formal languages, automata, and complexity*. Benjamin-Cummings Publishing Co., Inc., 1989.

[Deh00] Patrick Dehornoy. *Mathématiques de l'informatique : cours et exercices corrigés*. Dunod, 2000.

[HHF85] Douglas R Hofstadter, Jacqueline Henry, and Robert French. *Gödel, Escher, Bach : les brins d'une guirlande éternelle*. InterEditions, 1985.

[Wol06] Pierre Wolper. *Introduction à la calculabilité*. Dunod, 2006.