LA THÉORIE EON

Patrick Bellot

Télécom Paris
Institut Polytechnique de Paris
bellot@telecom-paris.fr

MITRO 202

Citons :

FOUNDATIONS OF CONSTRUCTIVE MATHEMATICS
M.J. Beeson
Metamathematical Studies, Springer Verlag, 1985

et

INTUITIONNISM, AN INTRODUCTION
A. Heyting
Studies in Logic, North Holland, 1956

et

INTRODUCTION TO METAMATHEMATICS
S.C. Kleene
Biblioteca Mathematica, North Holland, 1952

La notion de preuve

On rappelle que la notion de preuve est au centre de la logique intuitionniste. En cela, elle se distingue de la logique classique pour qui la notion de vérité est beaucoup plus importante.

La théorie de la réalisabilité de Kleene fait le lien entre l'aspect constructif de la logique intuitionniste et la notion de processus calculable.

Elle permet, entre autres, d'extraire des programmes à partir de preuves d'existence. On démontre qu'un objet existe et de la preuve, on extrait un programme qui calcule cet objet.

Mais il y a d'autres utilisations...

La notion de réalisabilité

La théorie de la réalisabilité associe des objets aux formules prouvées. Ces objets sont appelés des réalisations.

Ces objets sont en général construits par induction structurelle tout au long de la preuve de la formule.

Exemple.
Gödel avait associé des entiers aux formules prouvables, ce sont le fameux codage de Gödel.

Dans ce chapitre, nous allons nous servir d'un exemple de réalisabilité à travers la théorie EON : *Elementary theory of Operations and Numbers*

La notion de réalisabilité

On peut également associer, et ce n'est pas très différent, des objets de plus haut niveau comme des λ-expressions ou des termes de la Logique Combinatoire.

Le type et la forme des objets que l'on associe aux formules prouvées dépend de ce que l'on veut démontrer avec la réalisabilité.

C'est ce que nous allons faire avec la théorie EON de M. Beeson en associant aux formules prouvables des termes de la Logique Combinatoire.

Il est préférable d'utiliser les combinateurs plutôt que les λ-expressions afin de ne pas mélanger l'opérateur de liaison de variable λ avec les quantificateurs logiques ($\forall$ et $\exists$) qui lient également les variables. La définition de la substitution devient alors insurmontable.

Le langage d'EON

Les formules d'EON sont des formules du langage de la logique du premier ordre intuitionniste. Les termes d'EON sont les termes de la logique combinatoire.

En particulier la négation est définie par $\neg A \equiv_{def} (A \Rightarrow \bot)$ où $\bot$ est une contradiction, $S = K$ par exemple.

Dans cette section, on notera parfois l'application $(M\ N)$ sous la forme plus classique $M(N)$.

Les réalisations sont les termes de la Logique Combinatoire.

On supposera que, dans notre théorie des combinateurs, l'on a, en tant que primitives, les booléens, les entiers et leurs opérations, les opérations (π, π_1, π_2) sur les paires.

La partialité

Un terme M de la Logique Combinatoire peut avoir ou ne pas avoir de forme normale.

On introduit la notion « le terme M a une valeur » qui est une notion plus forte que celle de normalisabilité. On la note $M\downarrow$ et on la lit « le terme M a une valeur ».

Définition. $M\downarrow$ est définie par :

les constantes et les variables ont une valeur ;

si $(M\ N)\downarrow$ alors $M\downarrow$ et $N\downarrow$;

$(\lambda^{+}x\cdot M)\downarrow$;

si $M\downarrow$ et $N\downarrow$ alors $[M,N]\downarrow$.

Le système formel

Le système est un système de déduction naturelle intuitionniste où les preuves ont cette forme :

$$\frac{\vdots}{A}$$

avec éventuellement des hypothèses :

$$\frac{\dfrac{[A]}{A}}{A \Rightarrow A}$$

Les règles sont celles du chapitre « Formalismes logiques » à l'exception des règles $\exists_{intro}$ et $\forall_{elim}$.

Le système formel

On ne peut introduire un $\exists$ que si le témoin, ici M, a une valeur :

$$\frac{\begin{array}{c}\vdots \\ A(M) \wedge M\downarrow\end{array}}{\exists x \cdot A(x)}\ \exists_{intro}$$

On peut éliminer un $\forall$ uniquement avec un terme qui a une valeur :

$$\frac{\begin{array}{c}\vdots \\ [\forall x \cdot A(x)] \wedge M\downarrow\end{array}}{A(M)}\ \forall_{elim}$$

L'égalité et l'égalité faible

On ajoute à la logique un prédicat d'égalité tel que si $M = N$ alors M et N ont une valeur et ont la même valeur. On a l'axiome :

$$M\downarrow \ \Rightarrow M = M$$

On ajoute un prédicat d'égalité faible $M \simeq N$ qui est vraie si $M = N$ ou bien si ni M ni N n'ont de valeur. On admet l'axiome suivant:

$$M \simeq N \Rightarrow (A(M) \Rightarrow A(N))$$

Les entiers

On introduit un prédicat $M \in \mathbb{N}$ signifiant que M est un entier naturel.

On introduit la notation:

$$\forall x \in \mathbb{N} \cdot A(x) \equiv_{def} \forall x \cdot (x \in \mathbb{N} \Rightarrow A(x))$$

On se donne l'axiome de récurrence sur les entiers :

$$[A(0) \wedge \forall x \in \mathbb{N} \cdot (A(x) \Rightarrow A(x+1))] \Longrightarrow \forall x \in \mathbb{N} \cdot A(x)$$

On se donne l'axiome de cohérence :

$$\forall x \in \mathbb{N} \cdot \neg(x + 1 = 0)$$

La réalisabilité

On associe à toute formule A un autre formule $(e : A)$ de la même logique où e est une variable n'apparaissant pas dans la formule A.

$(e : A)$ est une *méta-notation* désignant une formule que l'on peut construire avec les définitions qui vont suivre.

On dit que e est la *réalisation* ou la *justification* de A.

Ce que signifie $(e : A)$ dépend de sa définition.

Si M est un terme alors $(M : A)$ représente la formule $(e : A)$ dans laquelle on a substitué M à e.

Définition de la réalisabilité

Soit A une formule et $e \notin FV(A)$, on définit $(e : A)$ par induction sur la formule A :

- si A est atomique alors $e : A \equiv_{def} A$;
- $e : A \wedge B \equiv_{def} \pi_1(e) : A \wedge \pi_2(e) : B$;
- $e : A \Rightarrow B \equiv_{def} \forall x \cdot [x : A \Rightarrow e(x)\downarrow \wedge e(x) : B]$;
- $e : \forall x \cdot A(x) \equiv_{def} \forall y \cdot [e(y)\downarrow \wedge e(y) : A(y)]$;
- $e : \exists x \cdot A(x) \equiv_{def} \pi_2(e) : A(\pi_1(e))$;
- $e : A \vee B \equiv_{def} \pi_1(e) \in \mathbb{N} \ \wedge [\pi_1(e) = 0 \Rightarrow \pi_2(e) : A; \pi_2(e) : B]$.

où $[A \Rightarrow B; C] \equiv_{def} [A \Rightarrow B] \wedge [\neg A \Rightarrow C]$ est une méta-notation.

La réalisabilité

Si l'on décide de prononcer $(e : A)$ comme

« e est une preuve intuitionniste de A »,

on retrouve approximativement la définition des preuves intuitionniste de Heyting-Kolmogorov déjà vue dans le chapitre *Introduction à la logique*.

On peut donc voir cette définition de la réalisabilité comme une manière d'*internaliser* la notion de preuve intuitionniste, c'est-à-dire de la décrire en utilisant le langage de la logique elle-même.

La réalisabilité

Définition.

Une formule A est *réalisable* s'il existe un terme M ayant les *mêmes* variables libres que A tel que l'on puisse prouver

$$M\downarrow \wedge M : A.$$

Le terme M est appelé une réalisation de A.

Exemple

Une fonction est parfois caractérisée par ses pré-condition et post-condition.

La pré-condition $P(x)$ doit être vérifiée par l'argument x de la fonction.

La post-condition $Q(x, y)$ doit être vérifiée par le résultat y et l'argument x de la fonction.

Exemple

$P(x) = "x \in \mathbb{N}"$ et $Q(x, y) = "y = 2x"$ caractérise la fonction $f(x) = 2x$.

On peut dire que la fonction est caractérisée par la formule :

$$\forall x \cdot P(x) \Rightarrow \exists y \cdot Q(x, y)$$

C'est la spécification de la fonction.

Exemple

Supposons qu'une spécification soit réalisable. On a :

$$
\begin{aligned}
& e : \forall x \cdot [P(x) \Rightarrow \exists y \cdot Q(x,y)] \\
\equiv\ & \forall x \cdot [e(x)\downarrow \wedge (e(x) : P(x) \Rightarrow \exists y \cdot Q(x,y))] \\
\equiv\ & \forall x \cdot [e(x)\downarrow \wedge \forall z \cdot [z : P(x) \Rightarrow e(x)(z)\downarrow \wedge e(x)(z) : \exists y \cdot Q(x,y)]] \\
\equiv\ & \forall x \cdot [e(x)\downarrow \wedge \forall z \cdot [z : P(x) \Rightarrow e(x)(z)\downarrow \wedge \pi_2(e(x)(z)) : Q(x,\pi_1(e(x)(z)))]]
\end{aligned}
$$

On remarque donc que s'il existe e réalisant la formule et si l'on se donne un x et une réalisation z de $P(x)$ alors $y = \pi_1(e(x)(z))$ vérifie $Q(x,y)$.

C'est cette propriété qui va être utilisée pour obtenir un programme à partir d'une preuve.

Réalisabilité et négation

Calculons $(e : \neg A)$ où A est une formule quelconque. On a :

$$
\begin{array}{lll}
 & e : \neg A & \\
\equiv & e : A \Rightarrow \bot & \\
\equiv & \forall x \cdot [x : A \Rightarrow e(x)\downarrow \wedge e(x) : \bot] & \\
\equiv & \forall x \cdot [x : A \Rightarrow e(x)\downarrow \wedge \bot] & \\
\Rightarrow & \forall x \cdot [x : A \Rightarrow \bot] & \text{car} \quad e(x)\downarrow \wedge \bot \Rightarrow \bot \\
\equiv & \forall x \cdot [\neg(x : A)] &
\end{array}
$$

Donc $e : \neg A \Rightarrow (\forall x \cdot \neg(x : A))$, c'est-à-dire que si A est réalisable, $\neg A$ ne peut pas l'être.

Formules négatives

Définition

Une formule *négative* est une formule ne contenant ni le connecteur de disjonction $\vee$ ni le quantificateur existentiel $\exists$.

Exemple

Par construction, toutes les formules $(e : A)$ sont négatives.

Rappel :

- si A est atomique alors $e : A \equiv_{def} A$;
- $e : A \wedge B \equiv_{def} \pi_1(e) : A \wedge \pi_2(e) : B$;
- $e : A \Rightarrow B \equiv_{def} \forall x \cdot [x : A \Rightarrow e(x)\downarrow \wedge e(x) : B]$;
- $e : \forall x \cdot A(x) \equiv_{def} \forall y \cdot [e(y)\downarrow \wedge e(y) : A(y)]$;
- $e : \exists x \cdot A(x) \equiv_{def} \pi_2(e) : A(\pi_1(e))$;
- $e : A \vee B \equiv_{def} \pi_1(e) \in \mathbb{N} \ \wedge [\pi_1(e) = 0 \Rightarrow \pi_2(e) : A; \pi_2(e) : B]$.

Réalisabilité et formule négative

Théorème

Une formule négative prouvable est réalisable.

Plu précisément, si A est une formule négative alors il existe un terme M ayant les mêmes variables libres que A et tel que

$$A \Rightarrow [M \downarrow \wedge (M : A)]$$

et

$$(Q : A) \Rightarrow A \text{ pour tout } Q.$$

Démonstration

La démonstration se fait par induction sur la structure de la formule (voir le poly ou bien Beeson 1985).

Exercice

Réaliser la démonstration ci-dessus.

Réalisabilité et prouvabilité

Théorème

Une formule prouvable sous des hypothèses réalisables est elle-même réalisable.

Démonstration

La démonstration se fait par induction sur la structure de la preuve intuitionniste de A. Elle est longue mais ne pose pas de difficultés particulières.

Application

Une formule prouvable est réalisable.

Démonstration

Une formule prouvable est prouvable sans hypothèse donc elle est prouvable sous des hypothèses réalisables.

Application

Une formule qui n'est pas réalisable n'est pas prouvable.

Réalisabilité et prouvabilité

Application

Une formule réalisable est cohérente avec la théorie.

Démonstration

Si elle n'était pas cohérente alors il existerait une preuve de la contradiction $\perp$ sous l'hypothèse A, donc une preuve de $\neg A \equiv A \Rightarrow \perp$. Mais si $\neg A$ est prouvable, alors elle est réalisable et l'on a vu que dans ce cas A ne pouvait pas être réalisable.

Propriété

Il existe des formules réalisables mais non prouvables.

(Beeson) D'après les résultats précédents, de telles formules doivent contenir soit le connecteur de disjonction, soit le quantificateur existentiel, soit les deux.
Par exemple, le principe de Markov :

$$[\forall n \in \mathbb{N} \cdot A(n) \vee \neg A(n)] \Rightarrow [\neg\neg\exists n \in \mathbb{N} \cdot A(n) \Rightarrow \exists n \in \mathbb{N} \cdot A(n)]$$

Exercice

Démontrer :

$$e : \forall n \in \mathbb{N} \cdot A(n) \Leftrightarrow \forall n \in \mathbb{N} \cdot [e(n) \downarrow \wedge e(n) : A(n)]$$

On rappelle : $\forall n \in \mathbb{N} \cdot A(n) \equiv \forall n \cdot [n \in \mathbb{N} \Rightarrow A(n)]$.

Résumé de la solution

$$
\begin{array}{ll}
 & e : \forall n \in \mathbb{N} \cdot A(n) \\
\equiv & e : \forall n \cdot [n \in \mathbb{N} \Rightarrow A(n)] \\
\equiv & \forall n \cdot [e(n)\downarrow \wedge [e(n) : n \in \mathbb{N} \Rightarrow A(n)]] \\
\equiv & \forall n \cdot [e(n)\downarrow \wedge \forall x \cdot [x : n \in \mathbb{N} \Rightarrow e(n)(x)\downarrow \wedge e(n)(x) : A(n)]]
\end{array}
$$

$$
\begin{array}{ll}
 & e : \forall n \in \mathbb{N} \cdot [e(n)\downarrow \wedge e(n) : A(n)] \\
\equiv & e : \forall n \cdot [n \in \mathbb{N} \Rightarrow [e(n)\downarrow \wedge e(n) : A(n)]] \\
\equiv & \forall n \cdot [e(n)\downarrow \wedge e(n) : [n \in \mathbb{N} \Rightarrow [e(n)\downarrow \wedge e(n) : A(n)]]] \\
\equiv & \forall n \cdot [e(n)\downarrow \wedge \forall x \cdot [x : n \in \mathbb{N} \Rightarrow e(n)(x)\downarrow \wedge e(n)(x) : [e(n)\downarrow \wedge e(n) : A(n)]]] \\
\equiv & \forall n \cdot [e(n)\downarrow \wedge \forall x \cdot [x : n \in \mathbb{N} \Rightarrow e(n)(x)\downarrow \wedge \\
 & \qquad \pi_1(e(n)(x)) : e(n)\downarrow \wedge \pi_2(e(n)(x)) : (e(n) : A(n))]] \\
\Leftrightarrow & \forall n \cdot [e(n)\downarrow \wedge \forall x \cdot [x : n \in \mathbb{N} \Rightarrow e(n)(x)\downarrow \wedge \\
 & \qquad e(n)\downarrow \wedge \pi_2(e(n)(x)) : (e(n) : A(n))]] \quad \text{car } e(n) \text{ est atomique} \\
\Leftrightarrow & \forall n \cdot [e(n)\downarrow \wedge \forall x \cdot [x : n \in \mathbb{N} \Rightarrow e(n)(x)\downarrow \wedge \\
 & \qquad \pi_2(e(n)(x)) : (e(n) : A(n))]] \\
\Leftrightarrow & \forall n \cdot [e(n)\downarrow \wedge \forall x \cdot [x : n \in \mathbb{N} \Rightarrow e(n)(x)\downarrow \wedge \\
 & \qquad e(n) : A(n)]] \quad \text{car } e(n) : A(n) \text{ est négative} \\
\Leftrightarrow & \forall n \cdot [e(n)\downarrow \wedge \forall x \cdot [x : n \in \mathbb{N} \Rightarrow e(n)(x)\downarrow \wedge e(n) : A(n)]]
\end{array}
$$

Exercice (sans solution)

Démontrer :

$$e : \exists n \in \mathbb{N} \cdot A(n) \Leftrightarrow \pi_1(e(n)) \in \mathbb{N} \wedge \pi_2(e(n)) : A(\pi_1(e(n)))$$

On rappelle : $\exists n \in \mathbb{N} \cdot A(n) \equiv \exists n \cdot [n \in \mathbb{N} \wedge A(n)]$.

Spécification d'une fonction

Un programme fonctionnel, de type LISP ou ML, peut être spécifié par deux prédicats. En effet, il associe un résultat y en fonction d'un argument x. La spécification prend la forme :

$$\forall x \cdot [P(x) \Rightarrow \exists y \cdot Q(x, y)]$$

et signifie que si l'argument x vérifie la propriété $P(x)$, la *pré-condition*, alors le résultat y existe et vérifie la propriété $Q(x, y)$, la *post-condition*.

Spécification d'une fonction

La division entière par 2 peut être spécifiée avec :

$$\begin{cases} P(x) & \equiv_{def} \quad x \in \mathbb{N} \\ Q(x,y) & \equiv_{def} \quad (x = 2 * y) \vee (x = 2 * y + 1) \end{cases}$$

soit

$$\forall x \cdot x \in \mathbb{N} \Rightarrow (x = 2 * y) \vee (x = 2 * y + 1)$$

La démonstration

Pour obtenir un programme calculant la fonction spécifiée, il faut démontrer la formule $\forall x \cdot [P(x) \Rightarrow \exists y \cdot Q(x, y)]$. On obtient alors une réalisation e de la formule. Cette réalisation vérifie :

$$\forall x \cdot [e(x) \downarrow \wedge \forall z \cdot [z : P(x) \Rightarrow e(x)(z) \downarrow \wedge \pi_2(e(x)(z)) : Q(x, \pi_1(e(x)(z)))]]$$

Le programme $f(x, z) = \pi_1(e(x)(z))$ prend en argument x et une réalisation de $P(x)$ et renvoie le résultat y qui vérifie $Q(x, y)$.

Le problème est évidemment ce z qui réalise $P(x)$. L'idéal est bien sûr que cet argument ne soit pas utilisé. On peut alors l'éliminer artificiellement. Dans le système CoQ de l'Inria, on distingue deux types de propositions : celles qui ont un contenu calculatoire et celles qui n'en ont pas. Si $P(x)$ n'a pas de contenu calculatoire, on peut négliger z qui ne sera pas utilisé dans les calculs.

Réalisation de l'axiome de récurrence

L'axiome de récurrence est :

$$[A(0) \wedge \forall x \in \mathbb{N} \cdot (A(x) \Rightarrow A(x+1))] \Longrightarrow \forall x \in \mathbb{N} \cdot A(x)$$

On se convaincra que la fonction R suivante st une bonne réalisation de l'axiome de récurrence.

$$R \equiv_{def} \lambda^+ e \cdot \lambda^+ x \cdot \lambda^+ z \cdot \left(\begin{array}{ll} \text{if} & x = 0 \\ \text{then} & \pi_1(e) \\ \text{else} & \pi_2(e)(x-1)(z)(R(e)(x-1)(z)) \end{array} \right)$$

où :

- $e : [A(0) \wedge \forall x \in \mathbb{N} \cdot (A(x) \Rightarrow A(x+1))]$,
- x doit être un entier naturel
- et $z : x \in \mathbb{N}$.

Exemple de synthèse de programme

On se sert de l'axiome de récurrence :

$$[A(0) \wedge \forall x \in \mathbb{N} \cdot (A(x) \Rightarrow A(x+1))] \Longrightarrow \forall x \in \mathbb{N} \cdot A(x)$$

avec :

$$A(x) \equiv \exists y \cdot (x = 2 \times y) \vee (x = 2 \times y + 1)$$

La preuve (presque formelle)

La base de l'induction : $A(0)$

De :

$\mathbf{K} : 0 = 2 \times 0,$

on déduit :

$[\mathbf{0}, \mathbf{K}] : (0 = 2 \times 0) \vee (0 = 2 \times 0 + 1),$

puis :

$[\mathbf{0}, [\mathbf{0}, \mathbf{K}]] : \exists y \cdot (0 = 2 \times y) \vee (0 = 2 \times y + 1).$

Soit :

$[\mathbf{0}, [\mathbf{0}, \mathbf{K}]] : A(0)$

La preuve (presque formelle)

Pas de récurrence : $\forall x \in \mathbb{N} \cdot A(x) \Rightarrow A(x+1)$

La formule s'écrit :

$\forall x \cdot x \in \mathbb{N} \Rightarrow [A(x) \Rightarrow A(x+1)]$

Supposons :

$\mathbf{e} : x \in \mathbb{N}$

et

$\mathbf{f} : A(x)$

On a :

$\mathbf{f} : \exists y \cdot (x = 2 \times y) \vee (x = 2 \times y + 1).$

Donc :

$\pi_2(\mathbf{f}) : (x = 2 \times \pi_1(f)) \vee (x = 2 \times \pi_1(f) + 1)$

La preuve (presque formelle)

On a

$$\pi_2(\mathbf{f}) : (x = 2 \times \pi_1(f)) \vee (x = 2 \times \pi_1(f) + 1)$$

Si :

$$\pi_1(\pi_2(\mathbf{f})) = 0$$

alors :

$$\pi_2(\pi_2(\mathbf{f})) : (x = 2 \times \pi_1(f))$$

sinon :

$$\pi_2(\pi_2(\mathbf{f})) : (x = 2 \times \pi_1(f) + 1)$$

Il a donc deux cas à traiter.

La preuve (presque formelle)

Supposons :

$\mathbf{u} : (x = 2 \times y)$

Alors :

$\mathbf{u} : (x + 1 = 2 \times y + 1)$

Puis :

$[\mathbf{1}, \mathbf{u}] : (x + 1 = 2 \times y) \vee (x + 1 = 2 \times y + 1).$

Et :

$[\mathbf{y}, [\mathbf{1}, \mathbf{u}]] : \exists y \cdot (x + 1 = 2 \times y) \vee (x + 1 = 2 \times y + 1)$

Donc :

$[\mathbf{y}, [\mathbf{1}, \mathbf{u}]] : A(x + 1)$

La preuve (presque formelle)

Supposons :

$\mathbf{u} : (x = 2 \times y + 1)$

Alors :

$\mathbf{u} : (x + 1 = 2 \times (y + 1))$

Puis :

$[\mathbf{0}, \mathbf{u}] : (x + 1 = 2 \times (y + 1)) \vee (x + 1 = 2 \times (y + 1) + 1)$

Et :

$[\mathbf{y} + \mathbf{1}, [\mathbf{0}, \mathbf{u}]] : \exists y \cdot (x + 1 = 2 \times y) \vee (x + 1 = 2 \times y + 1)$

Donc :

$[\mathbf{y} + \mathbf{1}, [\mathbf{0}, \mathbf{u}]] : A(x + 1)$

La preuve (presque formelle)

Des trois transparents précédents, on en déduit :

$$\left(\begin{array}{ll} \text{if} & \pi_1(\pi_2(\mathbf{f})) = \mathbf{0} \\ \text{then} & [\pi_1(\mathbf{f}), [\mathbf{1}, \pi_2(\pi_2(\mathbf{f}))]] \\ \text{else} & [\pi_1(\mathbf{f}) + \mathbf{1}, [\mathbf{0}, \pi_2(\pi_2(\mathbf{f}))]] \end{array}\right) : A(x+1)$$

Puis :

$$\mathbf{U_0} : A(x) \Rightarrow A(x+1)$$

avec :

$$\mathbf{U_0} \equiv_{def} \left(\lambda^+\mathbf{f} \cdot \begin{array}{ll} \text{if} & \pi_1(\pi_2(\mathbf{f})) = \mathbf{0} \\ \text{then} & [\pi_1(\mathbf{f}), [\mathbf{1}, \pi_2(\pi_2(\mathbf{f}))]] \\ \text{else} & [\pi_1(\mathbf{f}) + \mathbf{1}, [\mathbf{0}, \pi_2(\pi_2(\mathbf{f}))]] \end{array}\right)$$

La preuve (presque formelle)

Et enfin :

$$\mathbf{U} : \forall x \cdot x \in \mathbb{N} \Rightarrow [A(x) \Rightarrow A(x+1)]$$

c'est-à-dire :

$$\mathbf{U} : \forall x \in \mathbb{N} \cdot (A(x) \Longrightarrow A(x+1))$$

avec :

$$\mathbf{U} \equiv_{def} \left(\lambda^{+} x, z, f \cdot \begin{array}{ll} \textbf{Si} & \pi_1(\pi_2(f)) = 0 \\ \textbf{Alors} & [\pi_1(f), [1, \pi_2(\pi_2(f))]] \\ \textbf{Sinon} & [\pi_1(f) + 1, [0, \pi_2(\pi_2(f))]] \end{array} \right)$$

dans lequel x est l'entier naturel, z est la preuve que $x \in \mathbb{N}$.

La preuve (presque formelle)

On voit clairement que ce programme traite beaucoup d'informations inutiles.

Une solution est d'éliminer ces informations comme le font Nuprl, PX et CoQ.

L'autre solution est de considérer que la preuve de $x \in \mathbb{N}$ pour un entier naturel x est simplement K.

La preuve (presque formelle)

C'est l'axiome de récurrence permet de conclure

$$\mathbf{F} : \forall x \in \mathbb{N} \cdot \exists y \cdot (x = 2 \times y) \vee (x = 2 \times y + 1)$$

avec:

$$R \equiv_{def} \lambda^+ e \cdot \lambda^+ x \cdot \lambda^+ z \cdot \left(\begin{array}{ll} \text{if} & x = 0 \\ \text{then} & \pi_1(e) \\ \text{else} & \pi_2(e)(x-1)(z)(R(e)(x-1)(z)) \end{array} \right)$$

donc si $e = [\ [0, [0, K]]\ ,\ U\ ]$:

$$\mathbf{F} = \mathbf{R}(e)\lambda^+ x, z \cdot \left(\begin{array}{ll} \text{if} & x = 0 \\ \text{then} & [0, [0, K]] \\ \text{sinon} & \mathbf{U}(x-1)(z)(\mathbf{F}(x-1, K)) \end{array} \right)$$

et la fonction spécifiée est :

$$f(x) = \pi_1(F(x, K))$$

Le programme LISP décurryfié

```
(defun pi(x y) (cons x y))
(defun pi1(x) (car x))
(defun pi2(x) (cdr x))

(defun U(x z f)
  (if (eq (pi1 (pi2 f)) 0)
      (pi (pi1 f) (pi 1 (pi2 (pi2 f))))
      (pi (+ 1 (pi1 f)) (pi 0 (pi2 f)))
  )
)

(defun F(x z)
  (if (eq x 0)
      (pi 0 (pi 0 'K))
      (U (- x 1) z (F (- x 1) 'K))
  )
)

(defun d2(x) (pi1 (F x 'K)))
```

Le programme LISP décurryfié et optimisé

Ce type de programme est bien entendu inefficace mais il peut être amélioré en supprimant les informations logiques non indispensables au calcul, c'est le cas des arguments z dans nos programmes. On obtiendrait :

```
(defun U(x f)
  (if (eq (pi1 (pi2 f)) 0)
      (pi (pi1 f) (pi 1 (pi2 (pi2 f))))
      (pi (+ 1 (pi1 f)) (pi 0 (pi2 f)))
  )
)

(defun F(x)
  (if (eq x 0)
      (pi 0 (pi 0 'K))
      (U x (F (- x 1)))
  )
)

(defun d2(x) (pi1 (F x)))
```

Nota bene

Dans la réalisation de la règle de récursivité, on aurait pu choisir une boucle plutôt qu'une récursivité.

L'efficacité du programme obtenu dépend étroitement de la preuve utilisée pour le synthétiser. Une preuve mal tournée peut changer la complexité algorithmique du programme.

Même si ces programmes ne sont pas très efficaces, quand ils sont synthétisés formellement avec un système informatique, ils deviennent prouvés conformes à leurs spécifications.

Lien avec le langage PROLOG

L'exemple que nous avons développé dans ce chapitre, la division entière par 2, repose sur une spécification :

$$\forall x \cdot [x \in \mathbb{N} \Rightarrow \exists y \cdot ((x = 2 * y) \vee (x = 2 * y + 1))]$$

Or, une telle spécification est *directement exécutable* dans un PROLOG avec contraintes avec le code suivant :

```
f(x,y) :- x = 2*y .
f(x,y) :- x = 2*y + 1 .
```

Lien avec le langage PROLOG

Mais toutes les spécifications ne sont pas directement exécutables. Il suffit pour cela que l'espace de recherche du programme PROLOG soit infini. C'est le cas par exemple de les spécification du maximum de deux entiers comme étant le plus petit des majorants :

$$\forall x_1 \cdot \forall x_2 \cdot x_1 \in \mathbb{N} \land x_2 \in \mathbb{N} \\ \Rightarrow \exists y \cdot (\forall z \in \mathbb{N} \cdot (z \geqslant x_1 \land z \geqslant x_2 \Longrightarrow z \geqslant y))$$

Avec une telle spécification, PROLOG serait obligé de rechercher le plus petit des éléments dans un ensemble infini, celui des majorants des deux entiers et cela PROLOG ne sait pas le faire...

Conclusions

EON permet de synthétiser des programmes fonctionnels conformes à leurs spécifications grâce à une preuve d'existence.

Mais l'obtention de la preuve ne peut être automatisée. Cependant, une telle preuve peut être vérifiée automatiquement.

Les preuves en logique intuitionniste sont particulièrement difficile à obtenir.

Il faudrait également prouver la cohérence des spécifications.

N'oublions pas que la preuve influe sur la complexité du programme obtenu.