# Traduire l'univers des mathématiques en Dedukti, sans univers

Amélie Ledein, Elliot Butte

# Traduire l'univers des mathématiques en Dedukti, sans univers

Amélie Ledein[1*] et Elliot Butte[2]

[1] Université Paris-Saclay, Inria, CNRS, ENS Paris-Saclay,
Laboratoire Méthodes Formelles, France
amelie.ledein@inria.fr
[2] ENSIIE, France
elliot.butte@ensiie.fr

Ces dernières années, le nombre d'assistants à la preuve, de prouveurs automatiques et de vérificateurs de preuve n'a cessé de croître. L'un d'entre eux, le *framework* logique Dedukti, a pour objectif de rendre ces outils formels interopérables, c'est-à-dire rendre possible la réutilisation des preuves d'un outil $A$ dans un outil $B$. Un autre outil formel, Metamath, a la particularité d'être à la 4e place dans la liste des systèmes possédant le plus grand nombre de preuves parmi la liste des 100 théorèmes à prouver de Freek Wiedijk, tout en étant constitué de très peu de fonctionnalités : pas de type dépendant, pas de polymorphisme, ni de notion d'univers.

Afin d'agrémenter le nombre de preuves qu'il est possible de traduire dans Dedukti, nous présentons Mathiilde, un traducteur automatique de Metamath vers Dedukti. Comme ce traducteur peut utiliser deux encodages différents de Metamath vers Dedukti, nous discutons les avantages et les inconvénients du point de vue de l'interopérabilité.

## 1 Introduction

Dedukti [3], un *framework* logique basé sur le $\lambda\Pi$-calcul modulo théorie, a pour principal objectif l'interopérabilité en se voulant être un standard dans lequel il est possible de définir n'importe quelle logique, mais également avec lequel la traduction d'une logique vers une autre est plus aisée.

Deux cas d'étude sont basés sur une approche utilisant la théorie $\mathcal{U}$ [5], c'est-à-dire ayant pour objectif de simplifier la preuve pour qu'elle soit exprimable dans une logique plus faible, ici une extension, nommée STT$\forall$ [15], de la Théorie des Types Simples avec du polymorphisme prénexe et des opérateurs de type, afin de pouvoir la traduire plus facilement vers d'autres assistants de preuve. En effet, F. Thiré a traduit le petit théorème de Fermat [14, 15], écrit en Matita, tandis que Y. Géran a traduit le livre I des Éléments d'Euclide [2], écrit en Coq, vers HOL Light, Lean, Matita, OpenTheory (donc Isabelle/HOL et HOL 4), et PVS.

Une autre approche, utilisant l'alignement des concepts, a été suivie dans un autre cas d'étude. A. Assaf, R. Cauderlier et C. Dubois [4, 9] ont reconstruit au sein de Dedukti la preuve du Cribble d'Eratosthène initialement écrite en Coq, avec les entiers naturels de HOL. Pour ce faire, il a fallu démontrer des relations de morphisme entre les entiers de Coq et les entiers de HOL. Des travaux en cours s'intéressent également à l'alignement des concepts, c'est-à-dire à des procédés permettant d'identifier des concepts équivalents, comme par exemple les entiers naturels de HOL, de PVS ou encore de Coq, et ainsi éviter la redondance de ces concepts lors de la traduction.

*Financée par Digicosme.

Metamath [13] est un *framework* logique dans lequel de nombreux résultats mathématiques ont été formalisés, comme par exemple 74 problèmes sur les 100 faisant partie de la liste de Freek Wiedijk [1]. Ce résultat est particulièrement surprenant au vu du langage décrit dans le Metamath-book [13], puisque celui-ci ne permet de définir que des constantes, des axiomes et des preuves, sans type dépendant, ni polymorphisme, ni univers. Malheureusement, à notre connaissance, de très rares références existent à propos de Metamath [13, 10, 6, 7, 8], comme une formalisation en Metamath de la Matching Logic [10], mais ces références ne sont pas suffisantes pour formaliser ce *framework* logique. La référence principale est le Metamath-book [13] mais, par exemple, celui-ci ne présente pas de grammaire formelle du langage de Metamath, mais seulement une description en anglais.

Ainsi, la première contribution de cet article est constituée d'une proposition d'une grammaire formelle du langage de Metamath ainsi que d'une formalisation papier de l'étape de normalisation effectuée par au moins l'implémentation de référence d'un vérificateur de preuve pour Metamath. En plus de cette formalisation partielle de Metamath, cet article propose deux encodages de Metamath vers Dedukti, l'un plus proche de la philosophie de Metamath, l'autre plus proche des fonctionnalités courantes d'un assistant de preuve. La traduction automatique de ces encodages est en cours de développement [1], mais le prototype actuel permet déjà de récupérer une part non-négligeable de la bibliothèque standard de Metamath à l'aide de notre premier encodage, mais également environ 1 000 preuves écrites dans la logique propositionnelle à la Hilbert à l'aide du deuxième encodage. Ce prototype peut également être vu comme un nouveau vérificateur de Metamath écrit en OCaml, puisque la construction du $\lambda$-terme vérifié par la suite en Dedukti est similaire à la méthode de vérification de preuve effectuée par Metamath.

Après une brève présentation de Dedukti (Section 2), nous présentons plus en détails Metamath, notamment à l'aide d'un exemple d'axiomatisation et de preuve, mais également en expliquant la notion de portée et comment normaliser tout fichier Metamath (Section 3). Ensuite, nous proposons un premier encodage profond de Metamath en Dedukti, qui se veut être au plus proche de la philosophie même de Metamath (Section 4). De plus, nous proposons un deuxième encodage en Dedukti, plus lisible, qui est cette fois superficiel (Section 5). Ces deux encodages se basent sur une forme normalisée des fichiers Metamath, comme le montre la figure 1. Enfin, nous présentons les résultats de traduction de la bibliothèque standard de Metamath pour les différents encodages proposés (Section 6).

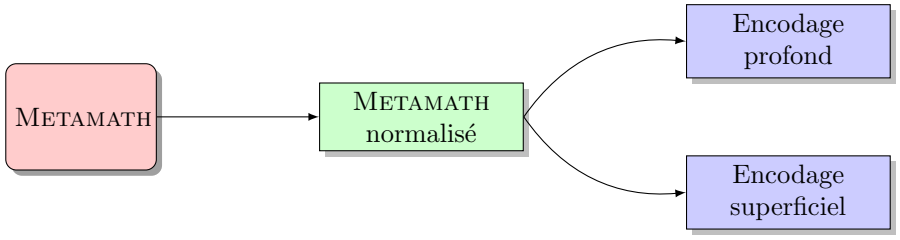


Figure 1 – Vue d'ensemble des deux traductions possibles de Metamath vers Dedukti

Dans la suite, les mots-clefs d'un langage ou ce qui est natif dans un langage seront distingués par de la couleur. Le langage de Dedukti se différenciera par une `couleur bleue`, tandis que le langage de Metamath sera différencié par une `couleur bordeaux`. Celles-ci facilitent la lecture, mais ne sont pas nécessaires à la compréhension.

1. https://gitlab.com/semantiko/MM2DK/translator

# 2 Dedukti

Dedukti [3] est un *framework* logique basé sur le $\lambda\Pi$-calcul modulo théorie, abrégé $\lambda\Pi\equiv_{\mathcal{T}}$, un $\lambda$-calcul avec types dépendants introduit par Cousineau et Dowek [11], et ayant la particularité de posséder une notion primitive de calcul définie à l'aide de règles de réécriture [12]. Plusieurs logiques ont déjà été encodées au sein de Dedukti, comme la logique des prédicats ou encore le Calcul des Constructions Inductives. Diverses logiques ont pu être encodées au sein de Dedukti, ce qui favorise la possibilité de rendre interopérables les preuves entre différents outils formels comme Coq ou PVS (Figure 2).

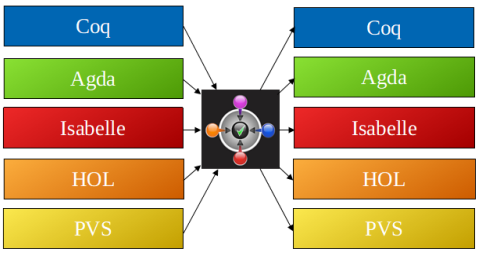


Figure 2 – L'interopérabilité des preuves grâce à Dedukti.

Dans cette section, nous ne présentons que les fonctionnalités disponibles dans Dedukti sur lesquelles nous nous appuyons dans la suite et qui permettront de faciliter, ultérieurement, les traductions vers d'autres formalismes.

**Typage et symboles.** La syntaxe du $\lambda\Pi\equiv_{\mathcal{T}}$ est directement accessible dans Dedukti : `TYPE` (sorte native), $\lambda$ (abstraction) et $\Pi$ (produit dépendant). Quand le produit dépendant $\Pi$ $(x : A)$, $B$ est en réalité non-dépendant, c'est-à-dire quand $x$ n'est pas une variable libre de $B$, nous notons $A \to B$.

La signature est définie à partir de symboles qui sont composés d'un nom et d'un type exprimable dans le $\lambda\Pi\equiv_{\mathcal{T}}$. Si la déclaration d'un symbole est faite avec le mot-clé `symbol` seul, on dit que le symbole est *défini*, sans aucune propriété particulière, alors qu'avec le mot-clé supplémentaire `constant`, le symbole est dit *constant* et ne peut pas être réduit par une règle de réécriture.
L'exemple suivant définit le type des entiers naturels ainsi que les deux constructeurs usuels.

```
1 constant symbol Nat : TYPE;
2 constant symbol 0 : Nat;
3 constant symbol S : Nat → Nat;
```

**Règles de réécriture.** Dans Dedukti, une règle de réécriture s'écrit `rule` *LHS* $\hookrightarrow$ *RHS* dans laquelle les variables libres sont notées `$x`, `$y`, etc. Il est possible d'y utiliser un joker (`_`) à gauche (*LHS*) lorsqu'une variable n'est pas utilisée dans la partie droite (*RHS*).
L'exemple suivant définit l'addition sur les entiers naturels définis précédemment.

```
4 symbol + : Nat → Nat → Nat;
5 rule $m + 0 ↪ $m;
6 rule $m + (S $n) ↪ S ($m + $n);
```

Les règles de réécriture autorisent l'ordre supérieur, peuvent être non linéaires et ne s'appliquent pas forcément en tête de terme, mais ne sont pas conditionnelles.

# 3 Metamath

L'objectif de cette section est de présenter Metamath [13]. Nous commençons par présenter comment définir une axiomatisation dans Metamath, ainsi que comment effectuer une preuve. Ensuite, nous expliquons qu'il est possible d'alléger la formalisation effectuée en Metamath à l'aide de la notion de portée, sans étendre l'expressivité de Metamath. Cette section présente également notre proposition de grammaire pour le langage de Metamath, ainsi que notre formalisation papier de l'étape de normalisation d'un fichier Metamath.

## 3.1 Écrire une axiomatisation

Le langage de Metamath permet de déclarer des constantes (`$c`), des variables (`$v`), le type de ces variables (`$f`), des contraintes sur ces variables (`$d`), ainsi que des hypothèses logiques (`$e`) associées soit à des axiomes (`$a`), soit à des théorèmes (`$p`). Les hypothèses commençant par `$f`, `$d` ou `$e` sont nommées respectivement *f-hypothèses*, *d-hypothèses* ou *e-hypothèses*. La figure 3 propose un extrait de la logique propositionnelle à la Hilbert définie en Metamath, où l'axiome `wi` correspond à la bonne formation de l'implication, et les axiomes `a2` et `mp` correspondent aux règles d'inférence suivantes :

$$\frac{}{\vdash ((\phi \texttt{ -> } (\psi \texttt{ -> } \chi)) \texttt{ -> } ((\phi \texttt{ -> } \psi) \texttt{ -> } (\phi \texttt{ -> } \chi)))}\ \texttt{a2} \qquad \frac{\vdash \phi \quad \vdash \phi \texttt{ -> } \psi}{\vdash \psi}\ \texttt{mp}$$

```
$c ( ) wff -> |- $.

${   $v φ ψ $.
     wph $f wff φ $.
     wps $f wff ψ $.
     wi $a wff ( φ -> ψ ) $. $}

${   $v φ ψ χ $.
     wph2 $f wff φ $.
     wps2 $f wff ψ $.
     wch  $f wff χ $.
     a2 $a |- ((φ -> (ψ -> χ)) -> ((φ -> ψ) -> (φ -> χ))) $. $}

${   $v φ ψ $.
     wph3 $f wff φ $.
     wps3 $f wff ψ $.
     min $e |- φ $.
     maj $e |- ( φ -> ψ ) $.
     mp  $a |- ψ $. $}
```

Figure 3 – Extrait de la logique propositionnelle à la Hilbert définie en Metamath

Certaines déclarations possèdent des labels, comme `wi`, `wch` ou encore `mp` : ceux-ci sont utilisés lors de l'élaboration de la preuve et sont donc uniques. D'un point de vue logique, une théorie définie en Metamath n'est donc composée que de constantes, d'axiomes et de théorèmes. Une constante correspond à un *token*, tandis que l'énoncé d'une hypothèse logique, d'un axiome ou d'un théorème est une liste non vide de *tokens*. Une variable peut être substituée par une liste non vide de *tokens*[2]. Ainsi, un fichier Metamath peut être vu comme un triplet $(\Sigma, \mathcal{A}, \mathcal{T})$, où la signature $\Sigma$ correspond à l'ensemble des constantes, $\mathcal{A}$ est l'ensemble des axiomes et $\mathcal{T}$ est l'ensemble des théorèmes.

La section suivante explique comment faire une preuve en Metamath.

2. Une option existe pour autoriser une variable à être substituée par aucun *token*, mais cette option est considérée *unsafe* par Metamath.

## 3.2 Faire une preuve

Le mécanisme de vérification de preuve de Metamath est basé sur la substitution de variables à l'aide d'une pile. La sous-section 3.2.1 illustre ce mécanisme sur un exemple, tandis que la sous-section 3.2.2 présente plus en détails les contraintes qu'il est possible d'exprimer sur les variables lors de la substitution.

### 3.2.1 Mécanisme général

Un théorème en Metamath est composé d'un énoncé, lui-même composé ou non de plusieurs hypothèses, et d'une preuve, c'est-à-dire d'une liste de labels. Un label est associé à une $f$-hypothèse, une $e$-hypothèse, un axiome ou un théorème. Par exemple, la preuve du théorème $\dfrac{\vdash (\phi \to (\psi \to \chi))}{\vdash ((\phi \to \psi) \to (\phi \to \chi))}$ `a2i`, écrite en Metamath, est disponible à la figure 4.

```
hyp $e  |- (φ -> (ψ -> χ)) $.
a2i $p  |- ((φ -> ψ) -> (φ -> χ)) $=
      wph ① wps wch ② wi ③ wi ④
      wph wps wi wph wch wi wi ⑤
      hyp ⑥
      wph wps wch a2 ⑦
      mp ⑧  $.
```

Figure 4 – Exemple de preuve écrite en Metamath

Sur le même principe que la notation polonaise inversée, le mécanisme de vérification de preuve de Metamath utilise une pile évitant toute ambiguïté sans avoir recours à l'utilisation de parenthèses au sein de la preuve. Les figures 5 à 12 illustrent ce mécanisme de vérification sur la preuve du théorème `a2i`. La pile de la figure 5 contient uniquement `wff` $\phi$, qui correspond au contenu associé au label `wph`. La pile de la figure 6 est obtenue en empilant du haut vers le bas les éléments `wff` $\psi$ et `wff` $\chi$, respectivement associés aux labels `wps` et `wch`. Ensuite, comme l'axiome `wi` possède deux variables, deux $f$-hypothèses lui sont associées, et donc deux éléments sont dépilés du bas vers le haut de la pile. Metamath cherche ensuite à unifier le premier élément récupéré de la pile avec la $f$-hypothèse `wps`, et le deuxième élément récupéré de la pile avec la $f$-hypothèse `wph`. La substitution $\{ \phi \mapsto \psi ; \psi \mapsto \chi \}$ est ainsi obtenue, et le nouvel élément `wff` $(\psi$ `->` $\chi)$ est mis sur la pile, comme le montre la figure 7. Le même procédé permet d'obtenir la pile de la figure 8, avec la substitution $\{ \phi \mapsto \phi ; \psi \mapsto (\psi$ `->` $\chi) \}$. La suite de labels `wph wps wi wph wch wi wi` permet de construire la pile de la figure 9 en empilant le nouvel élément `wff` $((\phi$ `->` $\psi)$ `->` $(\phi$ `->` $\chi))$. Ensuite, le label `hyp` met le contenu de l'hypothèse `hyp` sur la pile, comme le montre la figure 10. Enfin, l'élément en tête de la pile de la figure 11 est obtenu grâce à la suite de labels `wph wps wch a2`, et le label `mp` permet de finir la preuve, puisque le résultat de l'unification est syntaxiquement égal au contenu associé au label `a2i`, comme le montre la figure 12.

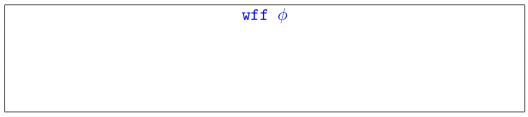


Figure 5 – État de la pile en ①

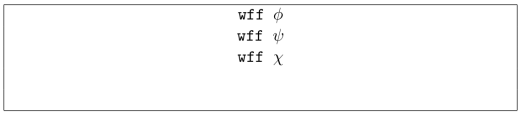


Figure 6 – État de la pile en ②

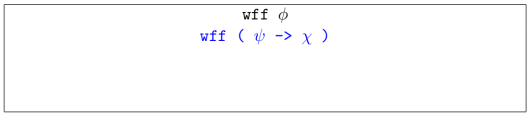


Figure 7 – État de la pile en ③

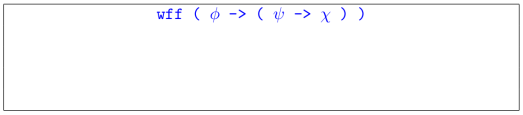


Figure 8 – État de la pile en ④

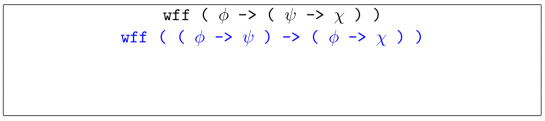


Figure 9 – État de la pile en ⑤

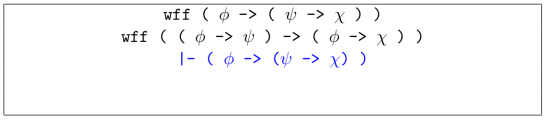


Figure 10 – État de la pile en ⑥


Figure 11 – État de la pile en ⑦

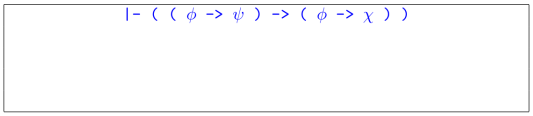


Figure 12 – État de la pile en ⑧

La sous-section suivante explique comment sont utilisées les déclarations imposant des contraintes sur les substitutions de variables (`$d`), lors du processus d'unification.

### 3.2.2 Une fonctionnalité un peu particulière

La section 3.1 mentionnait des déclarations imposant des contraintes sur les substitutions de variables qu'il est possible d'effectuer. Ces déclarations commencent par la balise `$d` et ont la sémantique suivante : `$d x y $.` signifie que pour toute substitution $[x \mapsto t_1, y \mapsto t_2, \dots]$ :

— $t_1$ et $t_2$ n'ont pas de variable en commun

— Pour toutes variable $v$ de $t_1$ et variable $w$ de $t_2$, `$d v w $.` est vrai dans le contexte local.

Cette fonctionnalité permet par exemple de modéliser le $\lambda$-calcul comme écrit ci-dessous, où `Subst l y v l'` signifie que `l'` correspond à la substitution de `y` par `v` dans `l`.

```
$c term var lam app Subst $.
$v x y l l' v $.
vx $f var x $.      vy $f var y $.
tl $f term l $.     tb $f term l' $.     tv $f term v $.
$d x y $.      $d x v $.
he $e Subst l y v l' $.
as $a Subst (lam x l) y v (lam x l') $.
```

Aucune indication n'existe dans la preuve pour vérifier ces contraintes. Metamath vérifie en interne les contraintes imposées par ces déclarations, à chaque fois qu'il effectue une substitution.

### 3.3 Utiliser la notion de portée

Le langage de Metamath offre un peu plus de souplesse que ce que nous avons pu présenter à la section précédente. En effet, il est possible d'importer un fichier Metamath dans un autre fichier à l'aide de la commande `$[` *filename* `$]`, mais il est également possible de jouer sur la portée des hypothèses pour éviter des redondances (`${` *section* `$}`). L'exemple de la figure 3 est très verbeux car il nécessite de recopier à de nombreuses reprises des déclarations de variables et d'hypothèses identiques (seul le label qui leur est associé est différent car tous les labels doivent être uniques). Plus concrètement, les fichiers A et B de la figure 13 sont définis sur la même signature $\Sigma \triangleq \{$ `(` ; `)` ; `nat` ; `0` ; `not` ; `=` ; `+` ; `*` ; `/` $\}$, et sont sémantiquement équivalents. En effet, le fichier A peut se lire ainsi :

Axiome `comm-plus` : Posons $a$ et $b$, des entiers naturels. Alors $a$ `+` $b$ = $b$ `+` $a$.

Axiome `eq-div` : Posons $a$, $b$, $c$ et $d$, des entiers naturels.
Si $c$ n'est pas égal à `0` et que $a$ `*` $b$ = $d$ `*` $c$, alors `(` $a$ `*` $b$ `)` `/` $c$ = $d$.

Le fichier B peut se lire ainsi :

Posons $a$, $b$, $c$ et $d$, des entiers naturels.

Axiome `comm-plus` : $a$ `+` $b$ = $b$ `+` $a$.

Axiome `eq-div` : Si $c$ n'est pas égal à `0` et que $a$ `*` $b$ = $d$ `*` $c$, alors `(` $a$ `*` $b$ `)` `/` $c$ = $d$.

Les fichiers A et B sont donc sémantiquement équivalents.

**Fichier A**

```
$c ( ) nat 0 not = + * / $.
${
      $v a b $.
      nata-p $f nat a $.
      natb-p $f nat b $.
      comm-plus $a a + b = b + a $.
$}

${
      $v a b c d $.
      nata-m $f nat a $.
      natb-m $f nat b $.
      natc-m $f nat c $.
      natd-m $f nat d $.
      notzero $e not ( c = 0 ) $.
      eq-mult $e a * b = d * c $.
      eq-div $a ( a * b ) / c = d $.
$}
```

**Fichier B**

```
$c ( ) nat 0 not = + * / $.
$v a b c d $.
nata $f nat a $.
natb $f nat b $.
natc $f nat c $.
natd $f nat d $.

comm-plus $a a + b = b + a $.

${
      notzero $e not ( c = 0 ) $.
      eq-mult $e a * b = d * c $.
      eq-div $a ( a * b ) / c = d $.
$}
```

Figure 13 – Deux exemples de fichiers Metamath

Il est également possible d'imbriquer des sections les unes dans les autres. La grammaire complète du langage Metamath que nous proposons est disponible à la figure 14. A notre connaissance, cette grammaire constitue la première formalisation du langage utilisé par Metamath. Celle-ci découle de notre lecture du Metamath-book [13], ainsi que de l'observation d'une multitude d'exemples.

| | | |
|---|---|---|
| Basic terminals | *carac* : := | *a-z* \| *A-Z* \| 0-9 |
| | *special-carac* : := | ! \| " \| # \| % \| & \| ' \| ( \| ) \| ∗ \| + \| , \| - \| . \| / \| : \| ; \| < \| = \| > \| ? \| @ \| [ \| \ \| ] \| ^ \| _ \| ` \| { \| "\|" \| } \| ∼ |
| | *white-space* : := | \| \t \| \r \| \n \| \f |
| | *text* : := | (*carac* \| *special-carac* \| *white-space* \| $)+ |
| | *filename* : := | ((*carac* \| *special-carac*)+).mm |
| | *math-symbol* : := | (*carac* \| *special-carac*)+ |
| | *variable* : := | *math-symbol* *(declared in variable symbol declaration)* |
| | *typecode* : := | *math-symbol* *(declared in constant symbol declaration)* |
| | *label* : := | (*carac* \| - \| _ \| .)+ |
| Comment | *comment* : := | $( *text* $) *(not nested)* |
| Included source file name | *import* : := | $[ *filename*+ $] |
| Constant symbol declaration | *c-decl* : := | $c *math-symbol*+ $. |
| Variable symbol declaration | *v-decl* : := | $v *math-symbol*+ $. |
| Block | *b-decl* : := | ${ (*comment* \| *v-decl* \| *b-decl* \| *hypothesis* \| *assertion*)∗ $} |
| Disjoint-variable restriction | *d-hypo* : := | $d *variable variable*+ $. |
| Variable-type hypothesis | *f-hypo* : := | *label* $f *typecode variable* $. |
| Logical hypothesis | *e-hypo* : := | *label* $e *typecode math-symbol*∗ $. |
| Axiomatic assertions | *a-assert* : := | *label* $a *typecode math-symbol*∗ $. |
| Provable assertions | *p-assert* : := | *label* $p *typecode math-symbol*∗ $= (*label* \| ?)∗ $. |
| File | *declaration* : := | *c-decl* \| *v-decl* \| *b-decl* |
| | *hypothesis* : := | *d-hypo* \| *f-hypo* \| *e-hypo* |
| | *assertion* : := | *a-assert* \| *p-assert* |
| | *file* : := | (*comment* \| *import* \| *declaration* \| *hypothesis* \| *assertion*)∗ |

FIGURE 14 – Grammaire de METAMATH

De plus, le langage de METAMATH possède très peu de sucre syntaxique. La déclaration $\texttt{\$c}\ c_1\ \ldots\ c_d\ \texttt{\$.}$ est équivalente aux déclarations $\texttt{\$c}\ c_1\ \texttt{\$.}$ ... $\texttt{\$c}\ c_d\ \texttt{\$.}$, de même que la déclaration $\texttt{\$v}\ v_1\ \ldots\ v_d\ \texttt{\$.}$ est équivalente aux déclarations $\texttt{\$c}\ v_1\ \texttt{\$.}$ ... $\texttt{\$c}\ v_d\ \texttt{\$.}$. La déclaration qui impose des contraintes sur les variables $\texttt{\$d}\ v_1\ \ldots\ v_b\ \texttt{\$.}$ correspond aux déclarations $\texttt{\$d}\ v_i\ v_j\ \texttt{\$.}$ pour tout $(i,j) \in [1; b-1] \times [i+1; b]$.

L'étude de l'exemple de la figure 13 illustre que deux fichiers METAMATH peuvent être sémantiquement équivalents mais syntaxiquement très différents. D'après le Metamath-book [13], une étape de normalisation est effectuée au moins par le vérificateur de référence de METAMATH, afin d'en simplifier la vérification. Cette étape de normalisation permet de réécrire le fichier B (Figure 13) et ainsi d'obtenir le fichier A (Figure 13), mais cette étape n'est pas formalisée dans le Metamath-book [13]. La section suivante propose une définition d'un fichier dit *normalisé*, tel que tout fichier METAMATH puisse se réécrire en un fichier normalisé.

## 3.4 Normaliser un fichier Metamath

Jusqu'à présent, nous avons illustré comment écrire une axiomatisation en METAMATH (Section 3.1) ainsi que comment faire une preuve (Section 3.2), mais également expliqué qu'il est possible d'écrire un fichier METAMATH moins verbeux à l'aide de la notion de portée (Section 3.3). Cette section donne un peu plus de sémantique à la syntaxe vue précédemment, en définissant ce qu'est un fichier METAMATH normalisé, puis en proposant une formalisation papier d'une transformation de tout fichier METAMATH en une version normalisée.

### 3.4.1 Définition d'un fichier normalisé

Par la suite, nous définissons ce qu'est un axiome normalisé, un théorème normalisé ainsi qu'un fichier METAMATH normalisé. Un *axiome normalisé*, une section constituée, dans l'ordre,

d'une liste de variables, d'une liste de $f$-hypothèses, d'une liste de contraintes sur ces variables, d'une liste de $e$-hypothèses et d'un énoncé. Un *théorème normalisé* est une section constituée, dans l'ordre, d'une liste de variables, d'une liste de $f$-hypothèses, d'une liste de contraintes sur ces variables, d'une liste de $e$-hypothèses, d'un énoncé et d'une preuve de cet énoncé. Un *fichier normalisé* est donc composé, dans l'ordre, d'une ou de plusieurs déclarations de constantes puis d'une liste d'axiomes normalisés ou de théorèmes normalisés.

Notre formalisation de la transformation d'un fichier METAMATH quelconque en une version normalisée est proposée à la section suivante.

### 3.4.2 Transformation en un fichier Metamath normalisé

Notre formalisation a besoin d'un contexte global noté $\Gamma = (\Sigma, \mathcal{A}, \mathcal{T})$, où $\Sigma$ correspond à la signature, $\mathcal{A}$ est l'ensemble des axiomes et $\mathcal{T}$ est l'ensemble des théorèmes. Nous utilisons également un contexte local noté $\Delta = [\ \Delta_0\ ;\ \ldots\ ;\ \Delta_{n-1}\ ;\ \Delta_n\ ]$ avec $\Delta_i = (\mathcal{V}_i,\ \mathcal{F}_i,\ \mathcal{D}_i,\ \mathcal{E}_i)$, où $\mathcal{V}_i$ est l'ensemble des variables à la profondeur $i$, $\mathcal{F}_i$ est l'ensemble des $f$-hypothèses à la profondeur $i$, $\mathcal{D}_i$ est l'ensemble des $d$-hypothèses à la profondeur $i$ et $\mathcal{E}_i$ est l'ensemble des $e$-hypothèses à la profondeur $i$. La notion de profondeur ici modélise la profondeur d'imbrication : une variable définie à la racine du fichier est à la profondeur 0, donc appartient à $\mathcal{V}_0$, tandis qu'une variable définie dans une section elle-même définie dans une section, sera à la profondeur 2, donc appartient à $\mathcal{V}_2$.

La formalisation que nous proposons est disponible à la figure 15, où $l \geq 0$, $d \geq 1$ et $i \geq 0$. Comme les constantes ne peuvent apparaître qu'à la profondeur 0 d'un fichier, leur traduction n'est valide que pour $i$ valant 0.

| | | |
|---|---|---|
| $\|\|\ s_1\ \ldots\ s_l\ \|\|_{norm}$ | $=$ | $\Gamma_{init}, \Delta_{init} \longmapsto \|\|\ s_1\ \|\|_0 \longmapsto \ldots \longmapsto \|\|\ s_l\ \|\|_0$ |
| $\|\|$ `$c` $c_1\ \ldots\ c_d$ `$.` $\|\|_0$ | $=$ | $\Gamma \overset{\Sigma}{\leftarrow} \Sigma \cup \{\ c_1\ ;\ \ldots\ ;\ c_d\ \},\ \Delta$ |
| $\|\|$ `$v` $v_1\ \ldots\ v_d$ `$.` $\|\|_i$ | $=$ | $\Gamma,\ \Delta \overset{\mathcal{V}_i}{\leftarrow} \mathcal{V}_i\ @\ [\ v_1\ ;\ \ldots\ ;\ v_d\ ]$ |
| $\|\|\ lab$ `$f` $t\ v$ `$.` $\|\|_i$ | $=$ | $\Gamma,\ \Delta \overset{\mathcal{F}_i}{\leftarrow} \mathcal{F}_i\ @\ [\ (lab,\ t,\ v)\ ]$ |
| $\|\|$ `$d` $d_1\ d_2$ `$.` $\|\|_i$ | $=$ | $\Gamma,\ \Delta \overset{\mathcal{D}_i}{\leftarrow} \mathcal{D}_i\ @\ [\ (d_1,\ d_2)\ ]$ |
| $\|\|\ lab$ `$e` $t\ e_1\ \ldots\ e_l$ `$.` $\|\|_i$ | $=$ | $\Gamma,\ \Delta \overset{\mathcal{E}_i}{\leftarrow} \mathcal{E}_i\ @\ [\ (lab,\ t,\ [e_1\ ;\ \ldots\ ;\ e_l])\ ]$ |
| $\|\|\ lab$ `$a` $t\ a_1\ \ldots\ a_l$ `$.` $\|\|_i$ | $=$ | $\Gamma \overset{\mathcal{A}}{\leftarrow} \mathcal{A}\ @\ [\ (\overline{\Delta}^{\leq i},\ lab,\ t,\ [a_1\ ;\ \ldots\ ;\ a_l])\ ],\ \Delta$ |
| $\|\|\ lab$ `$p` $t\ p_1\ \ldots\ p_l$ `$=` $l_1\ \ldots\ l_d$ `$.` $\|\|_i$ | $=$ | $\Gamma \overset{\mathcal{T}}{\leftarrow} \mathcal{T}\ @\ [\ (\overline{\Delta}^{\leq i},\ lab,\ t,\ [p_1\ ;\ \ldots\ ;\ p_l],\ [l_1\ ;\ \ldots\ ;\ l_d])\ ]),\ \Delta$ |
| $\|\|$ `${` $s_1\ \ldots\ s_d$ `$}` $\|\|_i$ | $=$ | $\Gamma,\ [\ \Delta_0\ ;\ \ldots\ ;\ \Delta_i\ ;\ \Delta_{emp}\ ] \longmapsto \|\|\ s_1\ \|\|_{i+1} \longmapsto \ldots \longmapsto \|\|\ s_d\ \|\|_{i+1}$ |

FIGURE 15 – Formalisation de l'étape de normalisation d'un fichier METAMATH

Initialement $\Gamma_{init} = (\emptyset,\ [\ ],\ [\ ])$ et $\Delta_{init} = [\ \Delta_0\ ]$ où $\Delta_0 = \Delta_{emp} \triangleq ([\ ],\ [\ ],\ [\ ],\ [\ ])$. Il faut préserver l'ordre des axiomes et théorèmes car leurs labels peuvent être utilisés dans l'élaboration des preuves des théorèmes, et nous avons vu que la vérification d'une preuve nécessite une pile dont l'ordre des éléments est important car l'inversion de deux hypothèses donnera un résultat tout à fait différent après substitution. Ainsi, nous utilisons rarement la structure mathématique d'ensemble car celle-ci ne préserve pas l'ordre des éléments. Nous optons plutôt pour la structure de liste, et notons $[\ ]$ la liste vide et $@$ l'opération de concaténation de deux listes.

Le résultat de notre traduction est un couple formé d'un contexte global et d'une liste de contextes locaux. La notation $T_1 \longmapsto T_2$ indique que la traduction $T_2$ débute avec le contexte

global et la liste de contextes locaux résultant de la traduction $T_1$. La notation $\Delta \overset{X_i}{\Leftarrow} Y$ indique que nous remplaçons la composante $X$ par $Y$ dans $\Delta_i$, c'est-à-dire dans le $i$-ème élément de la liste $\Delta$. Enfin, la notation $\overline{\Delta}^{\leq i}$ correspond à la concaténation, composante par composante, des $i$ premiers éléments de la liste $\Delta$. Les variables, ainsi que les $f$-hypothèses et $d$-hypothèses associées, ne sont gardées que si les variables apparaissent dans les $e$-hypothèses ou dans l'énoncé.

Le fichier normalisé correspond au contexte global $\Gamma_{final} \triangleq (\Sigma_{final}, \mathcal{A}_{final}, \mathcal{T}_{final})$ obtenu à la fin de la traduction. Il est presque possible d'obtenir un fichier Metamath valide en affichant successivement tous les éléments de $\Sigma_{final}$, puis de $\mathcal{A}_{final}$ et enfin de $\mathcal{T}_{final}$. Pour obtenir un fichier complètement valide, il faut assurer que tous les labels sont uniques, et donc procéder à certaines renommages lors de certaines déclarations, mais également dans les preuves. Nous n'avons pas formalisé ce renommage afin de ne pas alourdir notre formalisation.

Les deux encodages proposés par la suite se basent sur cette forme normalisée.

# 4 Un encodage profond de Metamath en Dedukti

Cette section propose un encodage profond de Metamath en Dedukti, c'est-à-dire un encodage au plus proche de la philosophie actuelle de Metamath.

## 4.1 Formalisation de l'encodage profond

La normalisation proposée à la section 3.4 nous indique que Metamath utilise deux méta-opérateurs afin de *lier* les variables, les différentes hypothèses et l'énoncé entre-eux : une quantification universelle, que nous notons $\forall_{MM}$, et une implication, que nous nous notons $\Rightarrow_{MM}$. Ces deux méta-opérateurs manipulent des *tokens*. Nous avons donc besoin de typer chacun des *tokens* :

```
7  symbol token : TYPE;
```

A présent, nous pouvons définir les méta-opérateurs notés $\forall_{MM}$ et $\Rightarrow_{MM}$ :

```
8  symbol ∀MM : (token → token) → token ;
9  symbol ⇒MM: token → token → token ;
10 notation ⇒MM infix right 10;
```

En Dedukti, il est usuel de définir un symbole permettant de passer d'une formule au type de ses preuves. Ici, ce symbole est noté `Prf` et est défini à la ligne 11.

```
11 injective symbol Prf : token → TYPE ;
```

Nous pouvons alors définir un *token* $\phi$, et une preuve du *token* $\phi$, notée $\psi$, comme ce qui suit :

```
12 symbol ϕ : token;
13 symbol ψ : Prf ϕ;
```

Il nous est maintenant possible d'expliciter un lien entre les méta-opérateurs de Metamath, et ceux de Dedukti, où `$ b.[a]` signifie que `a` peut apparaître dans `b` :

```
14 rule Prf (∀MM(λ a, $b.[a]) ) ↪ Π a, Prf $b.[a];
15 rule Prf ($a ⇒MM $b) ↪ Prf $a → Prf $b ;
```

Ensuite, nous définissons un opérateur de concaténation de *tokens* afin de pouvoir exprimer, par exemple, le contenu d'une hypothèse ou d'un énoncé.

```
16 symbol ++ : token → token → token;
17 notation ++ infix right 10;
18 rule ($a ++ $l) ++ $m ↪ $a ++ ($l ++ $m);
```

Afin d'alléger la traduction des preuves, nous déclarons le symbole # comme étant injectif afin de simplifier le processus d'unification effectué par Dedukti.

```
19 injective symbol # : token → token → token;
20 notation # infix right 10;
```

Ce symbole sera intercalé entre un *typecode* (Figure 14) et une liste de *tokens*.

**Traduire l'axiomatisation.** A l'aide des déclarations précédentes, voyons, sur un exemple, comment traduire une axiomatisation écrite en Metamath. Nous ne formalisons par cette traduction, par souci de légèreté.
L'axiomatisation disponible à la figure 3 est traduite en Dedukti ci-dessous. Les lignes 21 à 25 correspondent à la traduction de la signature, tandis que les lignes 26 à 28 correspondent à l'axiome `wi` et les lignes 29 à 31 à l'axiome du modus ponens.

```
21 constant symbol PL   : token;
22 constant symbol PR   : token;
23 constant symbol wff  : token;
24 constant symbol –>   : token;
25 constant symbol taquet : token;

26 constant symbol wi :
27   Prf (∀MM(λ ϕ, ∀MM(λ ψ, ( wff # ϕ ) ⇒MM( wff # ψ ) ⇒MM
28         ( wff # PL ++ ϕ ++ –> ++ ψ ++ PR ))));
29 constant symbol mp :
30  Prf (∀MM(λ ϕ, ∀MM(λ ψ, ( wff # ϕ ) ⇒MM( wff # ψ ) ⇒MM
31   ( taquet # ϕ ) ⇒MM( taquet # PL ++ ϕ ++ –> ++ ψ ++ PR ) ⇒MM( taquet # ψ ) ));
```

**Construire un $\lambda$-terme.** La construction d'un $\lambda$-terme suit le mécanisme de vérification des preuves Metamath. Les labels sont mis au fur et à mesure dans la pile en prenant au préalable un nombre d'éléments dans la pile égal au nombre de $f$-hypothèses et de $e$-hypothèses associées à l'énoncé du label. Nous n'écrivons pas explicitement les valeurs des variables, d'où l'utilisation de jokers, car Dedukti est capable de les inférer. Les lignes suivantes correspondent à la traduction de la preuve vue à la section 3.2. Le résultat de cette traduction est très similaire à la liste de labels initialement écrite en Metamath.

```
33 symbol a2i :
34   Prf (∀MM(λ φ, ∀MM(λ ψ, ∀MM(λ χ, ( wff # φ ) ⇒MM( wff # ψ ) ⇒MM( wff # χ ) ⇒MM
35     (taquet # PL ++ φ ++ –> ++ PL ++ ψ ++ –> ++ χ ++ PR ++ PR) ⇒MM
36       (taquet # PL ++ PL ++ φ ++ –> ++ ψ ++ PR ++ –> ++
37                         PL ++ φ ++ –> ++ χ ++ PR ++ PR ) ) ) ) ) :=
38   λ φ ψ χ wph wps wch hyp,
39     mp _ _
40        (wi _ _ wph (wi _ _ wps wch))
41        (wi _ _ (wi _ _ wph wps) (wi _ _ wph wch))
42        hyp
43        (a2 _ _ _ wph wps wch);
```

### 4.2 Premières conclusions liées à cet encodage

Ce premier encodage respecte profondément la philosophie propre de Metamath. Toutes les constantes jouent le même rôle, et le contenu des énoncés des hypothèses, des axiomes et des théorèmes sont des listes de *tokens*. Certains *tokens* ont du être renommés car certains caractères valides pour Metamath ne sont pas supportés par Dedukti comme `"`, |, `.` ou encore `(` et `)`. Finalement, cet encodage préserve la structure de liste initialement donnée par l'utilisateur, mais n'est très pas lisible pour un être humain, comme c'est le cas pour l'énoncé du théorème `a2i`. Ce premier encodage permet cependant de traduire un très grand nombre de preuves de la bibliothèque standard de Metamath, comme nous le verrons à la section 6.

Afin de gagner en lisibilité, nous proposons un second encodage qui tente de donner un peu plus de sémantique aux *tokens* à l'aide du typage. En Dedukti, ainsi que dans de nombreux autres outils formels, les éléments constituant la signature possèdent un nom et un type (et donc une arité). En Metamath, les éléments constituant la signature possèdent un nom et sont d'arité 0. La section suivante propose un encodage superficiel qui tente de trouver une structure d'arbre appropriée à partir de cette structure de liste. Cela revient à tenter de trouver les types des éléments constituant la signature (et donc leur arité).

## 5 Vers un encodage superficiel de Metamath en Dedukti

Cette section présente un encodage superficiel de Metamath en Dedukti dont l'objectif principal est de gagner en lisibilité en déterminant le type de chaque symbole de la signature.

### 5.1 La philosophie de notre encodage superficiel

L'idée générale est de considérer que les éléments de la signature qui sont des *typecodes* de $f$-hypothèses sont en réalité des types. Les axiomes qui ont pour *typecode* un type sont donc des axiomes de bonne formation et sont appelés des *axiomes syntaxiques*. Les autres axiomes sont appelés des *axiomes sémantiques*.

Il nous est donc possible de découper notre signature en trois sous-ensembles : l'ensemble des symboles qui sont des types, comme `wff`, l'ensembles des symboles qui apparaissent en tant que *typecode* mais qui ne sont pas des types, comme |−, et l'ensemble des symboles, nommés *opérateurs*, qui n'apparaissent jamais en tant que *typecode*, comme ->. Ces ensembles forment une partition disjointe de la signature initiale.

A partir de ces trois ensembles, nous donnons le type `TYPE` aux symboles de typage. Par exemple, la ligne suivante correspond à la traduction en Dedukti du *token* `wff` :
`constant symbol` wff `: TYPE ;`.

De plus, les axiomes syntaxiques nous permettent d'inférer le type des opérateurs. Par exemple, l'axiome `wi` nous permet d'inférer le type du symbole ->, et ainsi d'obtenir la traduction suivante en Dedukti :
`constant symbol` -> `:` wff → wff → wff `;`.

Cependant, il ne nous est pas toujours possible d'inférer toutes les informations voulues, comme par exemple le type du symbole |−. Cela fait l'objet de la sous-section suivante.

## 5.2 Quelques limites

Cette sous-section liste quelques difficultés rencontrées lorsque nous avons voulu traduire des fichiers Metamath en Dedukti avec notre second encodage.

**Inférer le type de certains symboles.** Certains symboles sont plus difficiles à typer que d'autres. C'est en particulier le cas des symboles qui apparaissent en tant que *typecode* mais qui ne sont pas des types, comme $|-$.

**Inférer le parsing des opérateurs.** Sous certaines hypothèses, il possible de se passer des informations de précédence et d'associativité, en connaissant, par exemple, le *token* associé à la parenthèse fermante et le *token* associé à la parenthèse ouvrante. Cependant, il est possible que le nom d'un opérateur soit composé de plusieurs *tokens*, comme par exemple l'opérateur de branchement `if-then-else` qui peut s'écrire avec trois *tokens* : `if _ then _ else _`. Ce cas de figure est très difficile à identifier syntaxiquement.

**Détecter le sous-typage.** Metamath permet de définir du sous-typage, comme le montre la déclaration suivante :

```
xnat $f nat x $.
xfloat $a float x $.
```

Cette déclaration indique que si `x` appartient à l'ensemble des entiers, alors `x` appartient aussi à l'ensemble des nombres à virgule. Il est possible traduire ce type de définition en Dedukti à l'aide d'une injection de l'ensemble de départ vers celui d'arrivée. Cette partie de la traduction n'a pas encore été implémentée à l'heure actuelle.

## 5.3 Une solution avec des annotations

Pour pallier aux problèmes liés à l'inférence de type et de parsing, il possible non plus de traduire un fichier Metamath, mais plutôt de traduire un fichier Metamath modulo d'autres informations fournies par l'utilisateur, à l'aide d'un fichier JSON par exemple.

A ce stade, nous proposons uniquement à l'utilisateur de préciser un nom de substitution valide pour Dedukti (`name_dedukti`), le type du symbole (`type`), la manière dont il faut le parser (`mixfix` et `precedence`) ou si le symbole est à considérer comme une parenthèse ouvrante ou fermante. Un exemple de fichier JSON est disponible à la figure 16 : celui-ci permet de traduire le symbole $|-$ vu précédemment.

```
{ name_metamath : "|−",
name_dedukti : "taquet",
  type : "wff → TYPE,"
   mixfix : "prefix",
    precedence : 40 }
          ⇓
constant symbol taquet : wff → TYPE ;
```

Figure 16 – Exemple de traduction d'un symbole à partir d'un fichier JSON

Ainsi, à l'aide du type précisé dans le fichier JSON, Dedukti est capable de vérifier le typage de la traduction de l'axiome du modus ponens : constant symbol mp : Π ($\phi$ : wff), Π ($\psi$ : wff), $|- \phi \to |- ( \phi \to \psi ) \to |- \psi$ ;.

De plus, la traduction d'une preuve Metamath en $\lambda$-terme est illustrée aux figures 17 à 24. Cette traduction suit le même mécanisme que celui que Metamath utilise pour vérifier une preuve. La subtilité est que c'est le contenu des $f$-hypothèses, des $e$-hypothèses et des axiomes syntaxiques qui est mis sur la pile, tandis que pour les axiomes sémantiques, nous utilisons le label qui lui est associé, en utilisant les arguments qui sont sur la pile. Ainsi, il est possible d'obtenir le $\lambda$-terme $\lambda\varphi$, $\lambda\psi$, $\lambda\chi$, $\lambda$`hyp`, `mp` $\alpha$ $\beta$ `hyp` (`a2` $\varphi$ $\psi$ $\chi$) avec $\alpha \triangleq$ ( $\varphi$ `->` ( $\psi$ `->` $\chi$ ) ) et $\beta \triangleq$ ( ( $\varphi$ `->` $\psi$ ) `->` ( $\varphi$ `->` $\chi$ ) ) à partir de la preuve de la figure 4.

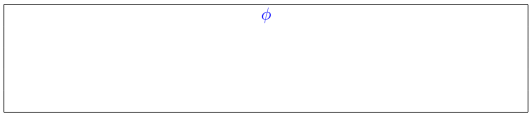


Figure 17 – État de la pile en ①

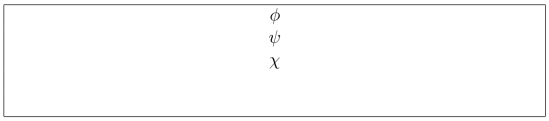


Figure 18 – État de la pile en ②

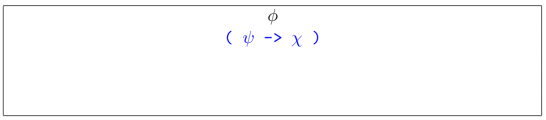


Figure 19 – État de la pile en ③

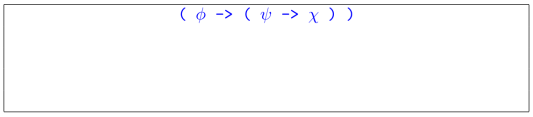


Figure 20 – État de la pile en ④

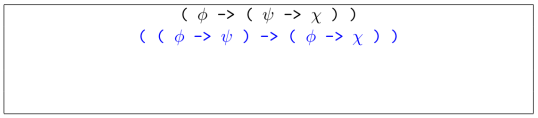


Figure 21 – État de la pile en ⑤

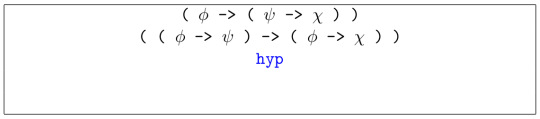


Figure 22 – État de la pile en ⑥

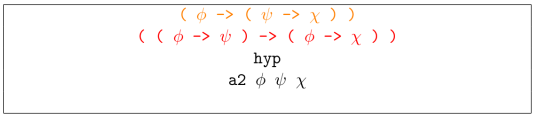


Figure 23 – État de la pile en ⑦

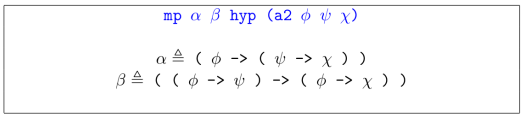


Figure 24 – État de la pile en ⑧

L'ajout d'un fichier JSON lors du processus de traduction semble être la seule manière de contourner les différents problèmes que nous rencontrons lorsque nous cherchons à obtenir un encodage superficiel de Metamath en Dedukti. Cependant, cette solution n'est pas complètement satisfaisante puisque la vérification d'un fichier Metamath ne dépend plus uniquement du fichier lui-même, mais également du fichier JSON fourni par l'utilisateur. Un fichier Metamath vérifié par Dedukti est donc correct modulo les informations fournies à l'aide du fichier JSON donné par l'utilisateur lors de la traduction. Une question ouverte résiste donc : quelle(s) propriété(s) les informations données dans le fichier JSON doivent-elles vérifier pour que la traduction soit pertinente et correcte ?

# 6 Traduction de la bibliothèque standard de Metamath

Cette section étudie la traduction de la bibliothèque standard de METAMATH avec nos deux encodages, afin de valider les encodages proposés. Après avoir présenté les résultats obtenus, nous proposons une extension commune à l'encodage superficiel et l'encodage profond.

## 6.1 Résultats de la traduction pour chaque encodage

La bibliothèque standard de METAMATH est composée de neuf fichiers disponibles sur GitHub : https://github.com/metamath/set.mm. Les résultats obtenus avec les implémentations actuelles des traductions utilisant nos deux encodages sont disponibles à la figure 25.

| | demo0.mm | miu.mm | peano.mm | big-unifier.mm | hol.mm | ql.mm | nf.mm | iset.mm | set.mm |
|---|---|---|---|---|---|---|---|---|---|
| **Taille du fichier** | 1.4K | 4.6K | 28K | 39K | 231K | 1.8M | 8.3M | 11M | 273M |
| **Nombre de** : | | | | | | | | | |
| - déclaration(s) | 19 | 27 | 116 | 29 | 2 768 | 936 | 24 967 | 30 813 | 196 976 |
| - axiome(s) | 7 | 10 | 48 | 4 | 77 | 71 | 363 | 467 | 2711 |
| - preuve(s) | 1 | 1 | 0 | 2 | 1 138 | 138 | 5 966 | 8 990 | 38 766 |
| **Pourcentage traduit avec l'encodage** : | | | | | | | | | |
| - profond | 100 | 0 | 100 | 100 | 95,87 | 100 | 79,07 | 80,29 | 65,26* |
| - profond étendu | 100 | 0 | 100 | 100 | 100 | 100 | 100 | 100 | 100* |
| - superficiel | 100 | 0 | 0 | 100 | 0 | 0+ | 0+ | 0+ | 2,52+ |

FIGURE 25 – Pourcentage de traduction de la bibliothèque standard de METAMATH pour les différents encodages proposés

Les pourcentages obtenus indiquent la proportion traduite pour chaque fichier et pour chaque encodage. Les fichiers obtenus ont pu être vérifiés par DEDUKTI, sauf pour les pourcentages ayant une étoile (*). Le fichier `set.mm` étant très volumineux, nous n'avons pas encore réussi à vérifier le typage de la totalité des deux fichiers de traduction obtenus, alors que METAMATH n'a besoin que de quelques secondes pour vérifier le fichier `set.mm`. Cette différence s'explique en partie car notre traducteur ne fonctionne qu'avec des fichiers METAMATH décompressés. Par exemple, la taille du fichier `set.mm` compressé est de 40.8M, et il faut plusieurs heures pour décompresser ce fichier. De plus, nous n'avons pas pu traduire le fichier `miu.mm` car celui-ci autorise la substitution d'une variable par *rien*, fonctionnalité considérée *unsafe* par METAMATH lui-même.

Dans le cas de l'encodage profond présenté à la section 4, la quasi-totalité des axiomatisations et des preuves a pu être traduite. Les preuves non traduites utilisent des variables libres, ce qui est possible car METAMATH fait l'hypothèse que tous les types qu'il manipule sont non vides. Les fichiers `hol.mm`, `nf.mm`, `iset.mm` et `set.mm` possèdent respectivement 47, 1 249, 1 772 et 13 466 preuves nécessitant la traduction de ce cas de figure. Nous avons donc implémenté un encodage profond étendu afin de pouvoir traduire ces preuves : cette extension est suffisante pour traduire

toutes les preuves restantes, comme le montre la figure 25. Ce cas de figure est illustré dans la sous-section suivante, ainsi qu'une explication plus détaillée de l'extension codée pour traduire la totalité des preuves de la bibliothèque standard de Metamath avec notre encodage profond. Cette extension est également nécessaire pour notre encodage superficiel.

Dans le cas de l'encodage superficiel présenté à la section 5, nous nous sommes principalement concentrés sur la traduction de la logique propositionnelle à la Hilbert (extrait du fichier `set.mm`), dont nous avons réussi à traduire en Dedukti environ 1 000 preuves. Le fichier `peano.mm` n'a pas pu être traduit car nous n'avons pas encore implémenté la génération d'injections à partir des axiomes de sous-typage. Le fichier `hol.mm` n'a également pas pu être traduit car il nécessite la traduction d'un opérateur d'application n'ayant pas de *token*, comme le montre la figure 26.

```
$c term ( ) $.

$( Term variables $)
$v F T $.
tf $f term F $.
tt $f term T $.

$( A combination (function application). $)
kc $a term ( F T ) $.
```

Figure 26 – Extrait du fichier `hol.mm`

Le pourcentage de traduction des fichiers `ql.mm`, `nf.mm`, `iset.mm` et `set.mm` est certainement plus élevé que celui annoncé car nous n'avons pas cherché à écrire la totalité des fichiers JSON nécessaires. En effet, cette tâche est longue et fastidieuse car il faut réussir à inférer manuellement un parsing et un typage correcte pour tous les *tokens* dont cela est nécessaire. Il nous semble plus judicieux d'essayer de trouver une méthode automatique pour trouver ces informations, vérifier la conformité du fichier traduit en Dedukti, puis essayer une nouvelle possibilité de parsing ou de typage en cas d'échec.

## 6.2 Une extension commune pour chaque encodage

Certaines preuves utilisent des variables qui ne sont pas liées dans l'énoncé du théorème. Cela est possible car, implicitement, Metamath fait l'hypothèse que tout type est habité : il existe donc au moins une variable pour chaque type. Cependant, ces preuves utilisent des variables libres, qui n'existent plus lors de l'étape de normalisation. Considérons la formalisation en Metamath suivante afin d'illustrer nos propos :

```
$c ( ) -> <-> wff |- $.
$v ph ps $.
wph $f wff ph $.
wps $f wff ps $.
wi $a wff ( ph -> ps ) $.

id $p |- ( ph -> ph ) $= ... $.

${    mpbir.min $e |- ps $.
      mpbir.maj $e |- ( ph <-> ps ) $.
      mpbir $p |- ph $= ... $.  $}
```

```
$c A. setvar class = T. $.
${    $v x $.
      vx.wal $f setvar x $.
      wal $a wff A. x ph $.  $}
${    $v x $.
      vx.cv $f setvar x $.
      cv $a class x $.  $}
${    $v A B $.
      cA.wceq $f class A $.
      cB.wceq $f class B $.
      wceq $a wff A = B $.  $}
wtru $a wff T. $.
```

```
${    $v x $.
      vx.tru $f setvar x $.
      df-tru $a |- ( T. <-> ( A. x x = x -> A. x x = x ) ) $.
      tru $p |- T. $=
            wtru vx.tru cv vx.tru cv wceq vx.tru wal vx.tru cv vx.tru cv
            wceq vx.tru wal wi vx.tru cv vx.tru cv
            wceq vx.tru wal id vx.tru df-tru mpbir $.  $}
```

La preuve du théorème `tru` utilise le label `vx.tru` introduisant la variable `x` dans la pile. Lors de la traduction vers Dedukti, la variable `x` sera libre car elle n'apparait pas dans l'énoncé du théorème `tru`. En effet, la forme normalisée de la section précédente est :

```
${    $v x $.
      vx.tru $f setvar x $.
      df-tru $a |- ( T. <-> ( A. x x = x -> A. x x = x ) ) $.  $}
tru $p |- T. $=
      wtru vx.tru cv vx.tru cv wceq vx.tru wal vx.tru cv vx.tru cv
      wceq vx.tru wal wi vx.tru cv vx.tru cv
      wceq vx.tru wal id vx.tru df-tru mpbir $.
```

Ainsi, la preuve du théorème `tru` utilise le label `vx.tru` qui n'est plus dans la portée du théorème `tru`. Nous devons étendre les encodages proposés afin de générer les habitants nécessaires pour chaque type. Dans le cadre de notre exemple, nous devons donc également générer le symbole suivant en Dedukti : `constant symbol` vxDottru `:` Prf ( setvar # x ) `;`.

Cette extension est suffisante pour traduire les preuves restantes de la bibliothèque standard de Metamath, dans le cas de notre encodage profond.

# 7 Conclusion

Nous venons de présenter deux encodages de Metamath vers Dedukti, l'un profond, l'autre superficiel.

L'encodage profond de Metamath en Dedukti nous apprend que la logique de Metamath est faible puisqu'elle n'est constituée que d'un opérateur de quantification et d'un opérateur d'implication. Cette logique semble suffisamment faible pour utiliser les travaux effectués par F. Thiré [14] afin de traduire le résultat obtenu vers Coq, Isabelle ou encore PVS.

L'encodage superficiel, plus lisible pour un être humain, permet de mettre en lumière les fonctionnalités usuelles modélisées dans une formalisation écrite en Metamath, comme le sous-typage. Cependant, cet encodage pose de nombreuses questions sur comment utiliser Dedukti pour traduire le résultat obtenu vers Coq, Isabelle ou encore PVS. De plus, il semble difficile de pouvoir complètement automatiser la traduction de fichiers Metamath dans cet encodage.

La traduction des $d$-hypothèses n'a jamais été présentée, car celles-ci ne sont utilisées que dans un mécanisme de vérification interne à Metamath dont aucune information n'apparaît dans la preuve écrite en Metamath. Il semble cependant possible d'étendre notre encodage profond pour également encoder cette vérification.

Cet article constitue donc une étape important pour permettre l'interopérabilité des preuves écrites en Metamath à l'aide de Dedukti, en utilisant l'approche basée sur la théorie $\mathcal{U}$ [5].

De plus, nous envisageons d'étendre le nombre de fichiers Metamath traduits, comme par exemple grâce à la formalisation en Metamath de la Matching Logic [10].

Enfin, M. Carneiro a proposé des modèles pour les différentes parties de la bibliothèque standard de Metamath [7]. Nous envisageons une comparaison plus approfondie de ces travaux avec les nôtres. M. Carneiro a également proposé Metamath-Zero [8], un outil formel inspiré de Metamath mais qui n'utilise pas de pile de preuves, et qui considère ses expressions comme des arbres et non comme des listes. Nous envisageons de mieux comprendre les liens entre Metamath et Metamath-Zero, afin d'évaluer la difficulté à récupérer des preuves de Metamath-Zero dans Dedukti.

**Remerciements** : Nous remercions Catherine DUBOIS, Chantal KELLER et Andrei PASKEVICH pour les remarques, commentaires et conseils qu'ils nous ont apportés tout au long de ce travail. Nous remercions également les reviewers anonymes pour leurs remarques et suggestions.

# Références

[1] Formalizing 100 Theorems - Freek Wiedijk. https://www.cs.ru.nl/~freek/100/.

[2] Translation of proofs of Euclid's book I into HOL Light, Lean, Matita, OpenTheory and PVS, note = https://github.com/karnaj/sttfa_geocoq_euclid.

[3] Ali Assaf, Guillaume Burel, Raphal Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant, and Ronan Saillard. Expressing theories in the λΠ-calculus modulo theory and in the Dedukti system. In TYPES : Types for Proofs and Programs, Novi SAd, Serbia, May 2016.

[4] Ali Assaf and Raphaël Cauderlier. Mixing HOL and Coq in Dedukti (Extended Abstract). In Cezary Kaliszyk and Andrei Paskevich, editors, Proceedings Fourth Workshop on Proof eXchange for Theorem Proving, PxTP 2015, Berlin, Germany, August 2-3, 2015, volume 186 of EPTCS, pages 89–96, 2015.

[5] Frédéric Blanqui, Gilles Dowek, Émilie Grienenberger, Gabriel Hondet, and François Thiré. Some axioms for mathematics. In Naoki Kobayashi, editor, 6th International Conference on Formal Structures for Computation and Deduction, FSCD 2021, July 17-24, 2021, Buenos Aires, Argentina (Virtual Conference), volume 195 of LIPIcs, pages 20 :1–20 :19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[6] Mario Carneiro. Conversion of HOL Light proofs into Metamath, 2014.

[7] Mario Carneiro. Models for Metamath, 2016.

[8] Mario Carneiro. Metamath Zero : The Cartesian Theorem Prover, 2019.

[9] Raphaël Cauderlier and Catherine Dubois. FoCaLiZe and Dedukti to the Rescue for Proof Interoperability. In Mauricio Ayala-Rincón and César A. Muñoz, editors, Interactive Theorem Proving, pages 131–147, Cham, 2017. Springer International Publishing.

[10] Xiaohong Chen, Zhengyao Lin, Minh-Thai Trinh, and Grigore Roşu. Towards a Trustworthy Semantics-Based Language Framework via Proof Generation. In Proceedings of the 33rd International Conference on Computer-Aided Verification. ACM, July 2021.

[11] D. Cousineau and Gilles Dowek. Embedding Pure Type Systems in the Lambda-Pi-Calculus Modulo. In TLCA, 2007.

[12] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In Handbook of Theoretical Computer Science, Volume B : Formal Models and Semantics, pages 243–320, 1990.

[13] Norman D. Megill. Metamath : A Computer Language for Mathematical Proofs. Lulu Press, Morrisville, North Carolina, 2019. http://us.metamath.org/downloads/metamath.pdf.

[14] François Thiré. Sharing a Library between Proof Assistants : Reaching out to the HOL Family. Electronic Proceedings in Theoretical Computer Science, 274 :57–71, jul 2018.

[15] François Thiré. Interoperability between proof systems using the logical framework Dedukti. (Interopérabilité entre systèmes de preuve en utilisant le cadre logique Dedukti). PhD thesis, École normale supérieure Paris-Saclay, Cachan, France, 2020.