

# Étude formelle de l'implémentation du code des impôts

Denis Merigoux, Raphaël Monat, Christophe Gaie

## ► To cite this version:

Denis Merigoux, Raphaël Monat, Christophe Gaie. Étude formelle de l'implémentation du code des impôts. 31ème Journées Francophones des Langages Applicatifs, Jan 2020, Gruissan, France. hal-02320347v3

**HAL Id: hal-02320347**

**<https://hal.inria.fr/hal-02320347v3>**

Submitted on 28 Dec 2019

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Étude formelle de l’implémentation du code des impôts\*

Denis Merigoux<sup>1</sup>, Raphaël Monat<sup>2</sup> et Christophe Gaie<sup>3</sup>

<sup>1</sup> Inria, Centre de Paris, 12 rue Simone Iff, 75012, Paris, France  
`denis.merigoux@inria.fr`

<sup>2</sup> LIP6, Sorbonne Université – CNRS, 4 place Jussieu, 75005 Paris, France  
`raphael.monat@lip6.fr`

<sup>3</sup> Direction Générale des Finances Publiques, Bureau SI-1E, Le Montaigne, Noisy-Le-Grand  
`christophe.gaie@dgifp.finances.gouv.fr`

## Résumé

Le code des impôts définit dans son texte législatif une fonction mathématique permettant de calculer l’impôt sur le revenu d’un foyer fiscal. Afin de recouvrer l’impôt, cette fonction est implémentée sous la forme d’un algorithme par la Direction Générale des Finances Publiques (DGFIP), en utilisant un langage dédié appelé M (pour macro-langage). Nous proposons une sémantique formelle du langage M, testée grâce aux données publiées par la DGFIP. Cette formalisation, couplée à la publication par la DGFIP de la base de code M calculant l’impôt, nous donne accès à une formalisation complète de la portion du code des impôts définissant l’algorithme de calcul de l’impôt sur le revenu. Nous démontrons l’utilité d’une telle formalisation grâce à un prototype à base de solveurs SMT permettant d’inférer des méta-propriétés sur le calcul de l’impôt. Ces méta-propriétés peuvent ensuite compléter et affiner les analyses économiques existantes sur les effets redistributifs de l’impôt sur le revenu, mais aussi de diverses allocations. Plus généralement, une formalisation systématique des portions algorithmiques de la loi permettrait d’augmenter le niveau d’assurance sur la cohérence du système socio-fiscal français.

Trois artefacts logiciels accompagnent cet article : une formalisation mécanisée de la sémantique du langage M, un compilateur pour le langage M basé sur cette sémantique, ainsi que le prototype d’encodage du code des impôts dans le solveur SMT Z3.

## 1 Quelle implémentation pour le système fiscal français ?

Le principe de tout impôt est de prélever pour le compte de l’État une partie d’un flux monétaire ou d’un capital. Mais de quelle taille est cette partie ? Comment calculer le montant de l’impôt à partir des paramètres de l’individu, du ménage ou de l’entreprise qui y est soumise ? La réponse se trouve dans la loi, et en France tout particulièrement dans le code général des impôts. De ce fait, les lois définissant le montant de l’impôt évoquent des quantités chiffrables et mesurables : revenus, nombre de personnes à charge, etc. L’impôt est donc défini par une fonction mathématique  $f$ , qui associe à chaque ménage ou entreprise  $x$  (défini par les caractéristiques prévues dans la loi) un impôt unique  $f(x)$ .

Comment l’impôt est-il calculé en pratique ? Tout dépend de la complexité de la fonction  $f$ . Pour le cas d’impôts forfaitaires ou proportionnels à une somme d’argent,  $f$  est très simple et la valeur  $f(x)$  peut être calculée de tête. Mais le principe de progressivité de l’impôt sur le revenu impose une fonction  $f$  plus complexe ; il faut alors poser le calcul à partir de la déclaration de revenus. Avec l’arrivée de l’informatique, le calcul est automatisé et programmé

---

\*Ce travail a été en partie financé par le Conseil Européen de la Recherche dans le cadre de la bourse “Consolidator Grant” 681393 - MOPSA.

sur ordinateur ; il existe donc nécessairement un programme informatique, appelé  $P$ , dont la tâche est de calculer  $f(x)$  à partir de  $x$ .  $P$  et  $f$  sont équivalents dans un certain sens, puisqu'ils calculent la même quantité.

On reconnaît là une problématique récurrente des méthodes formelles : la correspondance entre une spécification ( $f$ ) et une implémentation ( $P$ ). Le but de cet article est donc d'appliquer certains résultats classiques du champ des méthodes formelles au couple  $f$  et  $P$  qui définit le calcul des impôts, mais aussi au reste du système socio-fiscal français : cotisations et allocations. Même si la méthode peut être appliquée à n'importe quel pays recouvrant des impôts, la France fait partie des pays où le calcul du montant de l'impôt est à la charge de la puissance publique à partir des déclarations fiscales. Il est donc crucial que le programme  $P$  soit la transcription exacte de la fonction  $f$  définie par la loi, sous peine de recours légaux des contribuables lésés par une erreur de calcul.

Les contributions de cet article sont les suivantes :

- une présentation et formalisation avec preuve de sûreté du typage de M, le langage utilisé par la Direction Générale des Finances Publiques (DGFIP) pour définir le programme  $P$  de calcul de l'impôt sur le revenu ;
- cette présentation est accompagnée d'une définition en Coq de la sémantique et d'une preuve de sûreté du typage, ainsi que d'un compilateur permettant d'exécuter le code du calcul des impôts ;
- un prototype basé sur un solveur SMT pour l'analyse de propriétés du système socio-fiscal.

La section 2 présente la vision ainsi que le cadre d'utilisation de M au sein de la DGFIP. La section 3 analyse le langage M sous un angle formel. Dans la section 4, on explore ensuite diverses pistes d'application découlant de la formalisation. Enfin, la section 5 remet ce travail dans le contexte des initiatives ayant pour objectif la formalisation de la loi.

## 2 Le langage M : historique et utilisation à la DGFIP

### 2.1 Présentation du calcul de l'impôt sur le revenu

Le processus de gestion de l'impôt sur le revenu (noté dans la suite IR) nécessite de calculer le montant d'impôt d'une situation fiscale dans trois grands types de situations :

- le calcul primitif permet de connaître le montant d'impôt dû sur les revenus de l'année précédente. Cette modalité de calcul peut être utilisée pour réaliser une simulation d'impôt, pour informer le contribuable ou pour fixer le montant d'impôt dû par le contribuable (en termes légaux, pour “figer le montant d'impôt dû par le contribuable dans le cadre du rôle<sup>1</sup> d'imposition à l'impôt sur le revenu”).
- le calcul correctif permet de modifier le calcul primitif. Cette modalité peut être notamment utilisée lors d'un contentieux au bénéfice ou au détriment d'un contribuable. Il s'applique aux revenus déclarés pour l'année  $N - 1$ .
- le calcul de modulation permet de recalculer le taux de prélèvement à la source en cours d'année. Ce calcul fait généralement suite à une modification de situation de famille (mariage, divorce, naissance, décès, ...) ou à une évolution des revenus du foyer fiscal (perte d'emploi, reprise d'activité, nouveaux revenus fonciers, ...). Cette modalité s'applique aux revenus contemporains de l'année  $N$  (vision prospective des revenus qui seront déclarés). Un dispositif anti-abus est également prévu.

---

1. Le rôle d'imposition est le titre exécutoire en vertu desquels les comptables publics effectuent et poursuivent le recouvrement de l'impôt sur le revenu.

Au vu des différentes modalités de calcul, mais également de la diversité des applications qui doivent les réaliser, la « calculette IR » a été créée en 1990. Elle permet d'assurer l'unicité et la cohérence du calcul par de multiples applications. Elle offre également un gain d'efficacité significatif puisqu'une seule équipe est en charge du développement de ce module qui gère plus de 4000 variables et plus de 1000 règles de taxation. Le code M publié comporte ainsi environ 92 000 lignes de code pour le millésime des revenus 2017. La réalisation d'un module de calcul commun en lieu et place d'un développement spécifique permet de réaliser un gain d'efficacité d'un facteur 5 environ (1 équipe réalise le travail de 6 ou 7 autres qui doivent néanmoins embarquer le composant mutualisé).

Voici quelques informations concernant la publication du code source :

- la première publication a eu lieu dans le cadre du hackathon #CodeImpot qui s'est déroulé en avril 2016<sup>2</sup> ;
- le dépôt officiel du code source est <https://framagit.org/dgfip/ir-calcul> ;
- ce dépôt dispose du code de taxation primitif des revenus 2010 à 2017 ;
- le prochain millésime de taxation (revenus 2018) contiendra les éléments ayant permis de calculer le taux de prélèvement à la source et de réaliser la modulation (en cas de changement de situation de famille et/ou de revenus).

## 2.2 Organisation des travaux de l'équipe en charge du calcul IR

Dès le mois de septembre, l'équipe développe simultanément le moteur de calcul primitif sur les revenus  $N - 1$  et celui pour la modulation des revenus  $N$ . Le choix d'implémentation retenu consiste à réutiliser le calcul de taxation pour la modulation. Seules les règles de gestion différentes sont réécrites. Ce fonctionnement permet de ne pas dupliquer le code dans un nouveau module de calcul et réduit donc l'effort de maintenance ainsi que les risques d'erreur.

Pour valider les développements, l'équipe s'appuie sur les jeux d'essai fournis par la maîtrise d'ouvrage<sup>3</sup>. Ceux-ci sont totalement fictifs<sup>4</sup> et permettent de valider les différentes règles de calcul. Les jeux s'enrichissent d'année en année, permettant ainsi d'augmenter la couverture des tests. Ils permettent en outre de réaliser des tests de non-régression dans une démarche de type intégration continue.

Le calendrier d'utilisation des différentes calculettes est illustré dans la figure 1. Quelques précisions :

- ILIAD est l'application de gestion de l'impôt sur le revenu dans les services des impôts des particuliers. Cette application est écrite en Oracle Forms.
- TélÉIR est l'application qui permet aux usagers de déclarer leurs revenus en ligne. Cette application est écrite en Java.
- GestPart est l'application de gestion des situations particulières. Elle permet notamment de corriger les déclarations des usagers qui seraient en anomalie via des actions d'agents DGFIP. Cette application est écrite en Java.
- EDI-IR est l'application qui permet aux comptables de procéder à la déclaration pour le compte de leurs clients, directement depuis leurs progiciels comptables par échange de données informatisées. Cette application est écrite en Java.

---

2. <https://www.etalab.gouv.fr/retour-sur-le-hackathon-codeimpot>

3. Pour le calcul primitif et le calcul de modulation, la maîtrise d'ouvrage est le bureau GF-1A en charge de l'animation de la fiscalité des particuliers. Pour le calcul correctif, c'est le bureau GF-1B en charge des applications d'assiette et de recouvrement forcé des impôts des particuliers.

4. Les jeux d'essais correspondent à des typologies de situations réelles mais aucune donnée n'est réelle puisque l'administration respecte absolument le secret fiscal garanti par la loi.

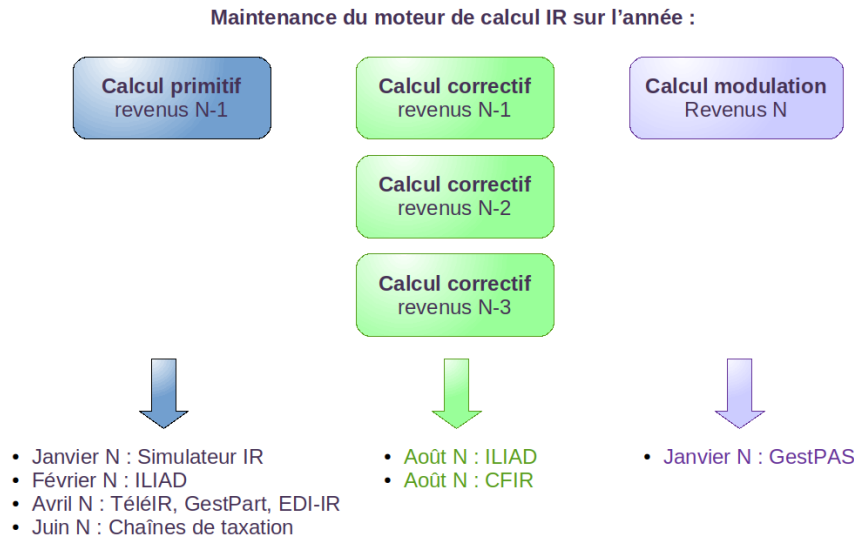


FIGURE 1 – Utilisations de la calculette IR

- Les chaînes de taxation sont les applications COBOL qui permettent notamment de calculer l'impôt sur le revenu, éditer l'avis d'imposition et établir le rôle d'imposition en vue de permettre le recouvrement.
- CFIR est l'application qui gère les conséquences financières à l'impôt sur le revenu. Elle permet le calcul et l'édition des droits et pénalités consécutives à des opérations de contrôle fiscal. Cette application est écrite en Visual Basic et Access.
- GestPAS est l'application qui permet d'effectuer les différentes opérations liées au prélèvement à la source. Cette application est écrite en Java.

Chaque application client embarque donc le moteur de calcul IR écrit dans le langage générique M. Ainsi la cohérence du calcul de l'impôt sur le revenu pour une version spécifique est garantie pour toutes les applications de la DGFIP. C'est en ce sens que nous pouvons dire que le langage M constitue un langage spécifique au domaine des finances publiques.

### 3 Étude formelle du langage M

Le calculateur du code des impôts écrit en langage M prend en entrée des paramètres numériques caractérisant un foyer fiscal au regard du fisc. Par exemple, la variable **1AJ** représente les revenus du premier déclarant, la variable **0AM** correspond au mariage d'un couple. Ces variables d'entrée sont soumises à des conditions de vérification dans le programme afin de modéliser des situations réalistes ; par exemple, **0AM** et **0AC** ne peuvent pas valoir 1 en même temps (on ne peut pas être marié et célibataire en même temps). Le calcul consiste ensuite en une suite d'affectation de variables (étant des scalaires ou des tableaux) à des expressions, et les opérations effectuées sont principalement des opérations arithmétiques, des conditionnelles et des calculs de seuil. Le langage ne possède pas de construction permettant d'exécuter des boucles de manière générale, il n'est donc pas Turing-complet. La sémantique de ce langage n'ayant pas été publiée par la DGFIP, nous avons dû faire un travail de rétro-ingénierie afin de l'établir. Par la suite, nous avons pu vérifier que notre sémantique était correcte en utilisant

la DGFIP comme oracle, ainsi qu'en utilisant certains fichiers de test. Le langage est assez dynamique, et cela est principalement dû à :

1. une valeur **indéfini** qui est utilisée pour dénoter les variables non initialisées ainsi que certaines situations souvent considérées comme des erreurs à l'exécution (telles que les divisions par zéro et les accès en dehors des bornes des tableaux) ;
2. cette valeur **indéfini** peut être convertie en des flottants via certaines règles de conversion (présentées en figure 6) ;
3. le langage possède des booléens mais en pratique, dans la base de code M publique, les opérations booléennes sont souvent réalisées à l'aide de l'arithmétique (0 pour faux, 1 pour vrai, l'addition comme « ou » logique, la multiplication comme « et », etc).

Nous commençons par présenter l'architecture générale du code de calcul de l'impôt, ainsi que la réduction vers un langage noyau légèrement simplifié, que nous appellerons par la suite «  $\mu M$  ». Nous présentons ensuite la syntaxe, le typage et la sémantique de ce langage noyau ; ces parties ont été formalisées en Coq. La dernière partie présente une implémentation *open-source* d'un compilateur pour le langage M d'origine, permettant de lire et exécuter le code de calcul mis à la disposition du public.

### 3.1 Réduction vers un langage noyau

Dans sa version du 6 octobre 2017, le code source du calculateur des impôts est séparé en 48 fichiers différents, qui totalisent 92 000 lignes. L'utilisation des variables est précédée d'une déclaration en tête de fichier, chaque variable étant associée à une description et un genre : une variable peut être « saisie » (c'est une entrée), « calculée », « restituée » (c'est une sortie). Des erreurs d'assertions, correspondant à des exceptions, sont aussi déclarées avec une description. Le code est ensuite séparé en des règles numérotées de manière unique (dont la numérotation n'influe pas l'ordre d'exécution). Chaque règle est annotée par une ou plusieurs applications, qui déclenchent son utilisation. Dans la suite, nous nous concentrons sur l'application *iliad*, présentée dans la section précédente. Les règles sont ensuite principalement des suites d'assignations, dont la syntaxe est décrite plus en détail en section 3.2. Les règles peuvent aussi contenir des assertions permettant de vérifier l'absence d'aberrations. La présentation effectuée ci-dessous se concentre sur le cœur du langage, omettant volontairement les facilités syntaxiques comme la notion de règle mentionnée précédemment, les boucles à itérations constantes utilisées pour définir les valeurs des tableaux, etc. De même, le cœur  $\mu M$  suppose que l'ordre d'exécution des assignations est fixé, tel que décrit dans le prochain paragraphe.

**Ordre d'exécution** Le code source ne présente pas d'ordre d'exécution apparent. L'outillage de la DGFIP semble faire une évaluation dynamique et paresseuse des variables en partant des variables de sorties qu'il faut calculer. Dans notre interpréteur OCaml, nous préférons exécuter une analyse de dépendance afin d'ordonner – partiellement – les assignations et assertions.

Avant d'exécuter un programme M, nous calculons donc le graphe de dépendance des variables, où  $x \rightarrow y$  si et seulement si la variable  $x$  est utilisée dans la définition de la variable  $y$ . Il est également nécessaire d'inclure dans ce graphe les nœuds correspondant aux assertions. Ce graphe sera parcouru lors de l'exécution dans l'ordre topologique qui permet de garantir qu'en l'absence de cycles dans le graphe, si  $x \rightarrow y$ , alors l'assignation de  $x$  sera évaluée avant celle de  $y$ . En présence de cycles, il est nécessaire de découper le graphe en chacune de ses composantes fortement connexes avant de les évaluer dans l'ordre topologique ; nous utilisons l'algorithme de Tarjan [15] pour identifier ces composantes en temps linéaire. L'évaluation d'une

composante fortement connexe comportant plusieurs variables définies circulairement est faite en évaluant chacune des variables séparément en supposant que la valeur des autres variables de la composante fortement connexe est égale à **indéfini** dans un premier temps. Il est alors possible d'effectuer plusieurs passes d'évaluation, les suivantes prenant comme valeurs d'entrée pour les variables de la composante fortement connexe les valeurs calculées au tour précédent. Le nombre de passages est choisi arbitrairement lors de l'exécution, et il semble être de l'ordre de 10 en pratique. Ce calcul itératif est fortement utilisé par la DGFIP pour encoder des calculs de majorations d'impôts ou de correctifs qui ont une structure itérative. Dans la suite, nous ignorons cette notion de calcul itératif sur les composantes fortement connexes, puisque le code de ces calculs peut être expansé. Le programme ne possède pas d'entrées ni de sorties en tant que telles dans ce cœur du langage : les variables d'entrée sont simplement définies comme égales à leur valeur d'entrée (ou à **indéfini** s'il n'y a pas de valeur d'entrée), et il suffit de lire la valeur des variables de sortie après exécution du programme.

### 3.2 Syntaxe du langage $\mu M$

La base du langage  $\mu M$  est la définition de variables qui se fait grâce à un langage d'expressions très simple dont la syntaxe est présentée en figure 2. La figure 3 donne un exemple de programme  $\mu M$  se conformant à cette syntaxe.

Un programme  $\langle \text{programme} \rangle$  est une liste de commandes  $\langle \text{commande} \rangle$ . Les commandes définissent la valeur des variables ainsi que des assertions conduisant à des erreurs lors de l'exécution du programme. Les variables peuvent être simples, ou bien correspondre à des tableaux de taille constante et connue à l'avance. Les valeurs simples sont des booléens ou des flottants. La valeur de chaque case du tableau est définie à l'aide d'une fonction d'un index générique **X**, variant entre les bornes de définition du tableau. Les expressions sont des variables (incluant l'index spécial de tableau appelé « **X** »), des littéraux, des combinaisons arithmétiques, logiques ou comparatives d'expressions, des conditionnelles, des appels de fonctions ou des accès dans des tableaux. Les fonctions sont fixées par le langage et ne peuvent pas être définies par un programme.

La fonction **arr** effectue un arrondi d'un flottant vers l'entier le plus proche, tandis que **inf** effectue un arrondi par défaut ; les fonctions **min** et **max** sont usuelles. La fonction **pos** (respectivement **pos\_ou\_nul**) renvoie 1 si son argument est strictement positif (respectivement positif ou nul), et 0 sinon. Les fonctions **present** et **null** sont utilisées pour discriminer la valeur **indéfini** des valeurs plus conventionnelles ; leur comportement est formellement défini dans la figure 6.

### 3.3 Typage du langage $\mu M$

Le typage du langage est défini dans la figure 4. Le but de ce système de type est d'assurer dans les programmes valides une distinction entre les variables scalaires et les variables de tableaux, ainsi que de séparer les types booléens et flottants.

**Typage des expressions** L'ensemble des variables est noté  $\mathcal{V}$ . Les types des valeurs sont  $T = \{\text{booléen}, \text{flottant}\}$ , tandis que les variables peuvent avoir un type scalaire simple ou un type tableau (ayant un contenu de type uniforme  $\tau \in T$ )  $\Theta = \{\text{scalaire}[\tau], \text{tableau}[\tau]\}$ . Le contexte de typage  $\Gamma$  est une fonction partielle  $\Gamma : \mathcal{V} \rightarrow \Theta$  mémorisant le type des variables. Les expressions arithmétiques, logiques ou de comparaison sont considérées comme des appels de fonctions afin d'alléger la présentation. Le type des fonctions est constant et défini dans un environnement global  $\Delta$  (présenté dans la figure 4). La majorité des fonctions admettent

$$\begin{aligned}
\langle \text{programme} \rangle &::= \langle \text{commande} \rangle \mid \langle \text{commande} \rangle ; \langle \text{programme} \rangle \\
\langle \text{commande} \rangle &::= \langle \text{var} \rangle := \langle \text{expr} \rangle \\
&\mid \langle \text{var} \rangle [ \text{X} ; \langle \text{entier} \rangle ] := \langle \text{expr} \rangle \\
&\mid \text{si } \langle \text{expr} \rangle \text{ alors } \langle \text{erreur} \rangle \\
\langle \text{expr} \rangle &::= \langle \text{var} \rangle \mid \text{X} \mid \langle \text{valeur} \rangle \\
&\mid \langle \text{expr} \rangle \langle \text{compop} \rangle \langle \text{expr} \rangle \mid \langle \text{expr} \rangle \langle \text{binop} \rangle \langle \text{expr} \rangle \mid \langle \text{unop} \rangle \langle \text{expr} \rangle \\
&\mid \text{si } \langle \text{expr} \rangle \text{ alors } \langle \text{expr} \rangle \text{ sinon } \langle \text{expr} \rangle \\
&\mid \langle \text{func} \rangle ( \langle \text{expr} \rangle, \dots, \langle \text{expr} \rangle ) \mid \langle \text{var} \rangle [ \langle \text{expr} \rangle ] \\
\langle \text{valeur} \rangle &::= \text{indéfini} \mid \langle \text{défini} \rangle \\
\langle \text{défini} \rangle &::= \langle \text{booléen} \rangle \mid \langle \text{flottant} \rangle \\
\langle \text{compop} \rangle &::= <= \mid < \mid > \mid >= \mid == \mid != \\
\langle \text{binop} \rangle &::= \langle \text{arithop} \rangle \mid \langle \text{logicop} \rangle \\
\langle \text{arithop} \rangle &::= + \mid - \mid * \mid / \\
\langle \text{logicop} \rangle &::= \&\& \mid \mid \mid \\
\langle \text{unop} \rangle &::= - \mid \sim \\
\langle \text{func} \rangle &::= \text{arr} \mid \text{inf} \mid \text{present} \mid \text{null} \\
&\mid \text{max} \mid \text{min} \mid \text{abs} \mid \text{pos} \mid \text{pos\_ou\_nul}
\end{aligned}$$
FIGURE 2 – Syntaxe du langage  $\mu M$ 

```

VAR0 = si (VAR1 > 0.0) alors - arr(VAR2 * 23 / 100) sinon 0.0;
TABLEAU[X; 10] = (3 * X * X + 2 * X + 1) * (1 - present(VAR1));

```

FIGURE 3 – Exemple de programme  $\mu M$ 

un type spécifique en entrée et en sortie, à l'exception des fonctions **null** et **present**, qui sont polymorphes en entrée.

A l'exception de T-UNDEF et de T-VAR-UNDEF, les règles de typage sont classiques : les valeurs booléennes sont des booléens, et de même pour les flottants. La règle T-UNDEF reflète la grande permissivité du langage lors de l'utilisation des **indéfinis**, puisque ceux-ci sont considérés comme des booléens ou des flottants. Le type d'une variable  $x$  est  $\tau$  si  $x$  est déclaré dans le contexte de typage comme étant un scalaire de type  $\tau$ . De manière similaire à T-UNDEF, T-VAR-UNDEF correspond à un comportement particulier du langage M : une variable qui n'est pas définie est évaluée en **indéfini**, qui a donc n'importe quel type. Le domaine de  $\Gamma$ ,  $\text{dom } \Gamma$ , est l'ensemble des variables définies dans le contexte  $\Gamma$ . La garde d'une conditionnelle doit être booléenne, et les expressions dans les branches de même type. L'accès dans un tableau est valide (pour le typage) dès lors que l'index est flottant. Un appel de fonction est bien typé si l'arité est correcte et si tous les arguments sont du type spécifié par une signature dans l'environnement global de typage des fonctions  $\Delta$ .

**Typage des commandes et programmes** Le typage des commandes transforme le contexte de typage au fil des assignations. La notation  $\Gamma[x \mapsto \theta]$  est le contexte  $\Gamma$  étendu avec l'annotation «  $x$  a le type  $\theta$  ». Dans le cas d'une assignation scalaire  $x := e$ , le contexte de typage est étendu d'une association entre  $x$  et **scalaire** $[\tau]$ , où  $\tau$  est le type de  $e$ . Dans le cas d'une assignation de tableau,  $x[X, n] := e$ , le contexte de typage est étendu pour lier  $x$  avec **tableau** $[\tau]$ , où  $\tau$  est le type de  $e$  dans l'environnement étendu où la variable générique d'index **X** est de type scalaire



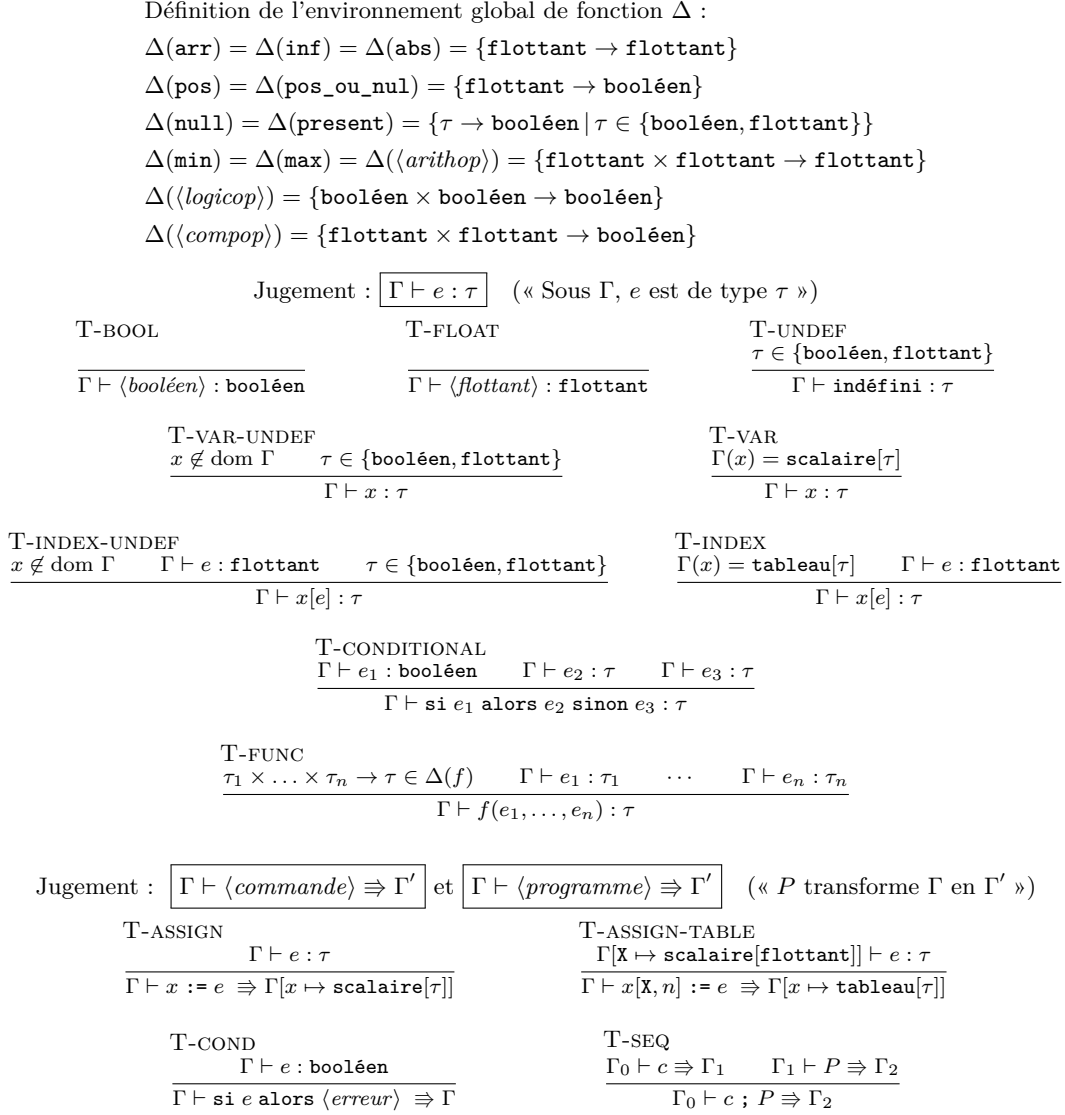


FIGURE 4 – Typage des expressions, commandes et des programmes

flottant. Les dernières règles sont classiques : les assertions doivent être de type booléen, et un programme est typé séquentiellement.

### 3.4 Sémantique du langage $\mu M$

**Sémantique des expressions** De manière générale, le langage  $M$  transforme les erreurs à l'exécution (telles que les divisions par zéro et les accès invalides dans les tableaux) en la valeur **indéfini** ; cette valeur peut ensuite être silencieusement convertie en d'autres valeurs, comme

l'illustre le comportement des opérateurs usuels en figure 6.

L'environnement mémoire d'exécution, noté  $\Omega$  dans la suite, est une fonction partielle des variables vers des valeurs scalaires (dénnotées dans la suite par  $v$ ) ou le contenu d'un tableau (noté dans la suite comme un  $n$ -uplet  $(v_0, \dots, v_{n-1})$ ). Les valeurs littérales sont évaluées directement en valeurs, et les variables sont définies par l'environnement mémoire  $\Omega$  (ou, à défaut, en la valeur **indéfini**). La variable spéciale d'indice de tableau **X** est considérée comme une variable normale. Les conditionnelles sont évaluées usuellement si la garde est évaluée en une valeur booléenne. Dans le cas où la garde est évaluée en **indéfini**, la conditionnelle est évaluée en **indéfini**. Si l'indice d'un tableau est évalué en **indéfini**, ou si l'indice est non-entier ou en dehors des bornes de définition, l'accès est évalué en **indéfini**. L'accès à un tableau avec un indice valide est traité de manière classique, en consultant l'environnement  $\Omega$ . Le comportement des fonctions est défini en figure 6, avec en particulier la spécification du comportement de la valeur **indéfini** pour chaque opérateur, qui est une des spécificités de M (les opérateurs unaires se comportent comme les opérateurs binaires similaires, et ils sont donc omis).

**Sémantique des commandes et programmes** L'environnement mémoire pour les commandes et les programmes est celui des expressions étendu pour supporter **erreur**, qui est utilisé lorsqu'une assertion est invalide. La règle D-ERROR stipule que si l'environnement **erreur** est rencontré, celui-ci est propagé tout au long du programme. L'environnement étendu est noté  $\Omega_c$ , il a pour valeur un environnement d'expression  $\Omega$  ou **erreur**. Les assignations de variables scalaires transforment l'environnement, de manière similaire à la règle de typage de l'assignation. Les assertions pouvant lever des erreurs sont exécutées de la manière suivante : la garde est évaluée en une valeur ; si elle vaut **vrai**, l'erreur est levée. Si elle vaut **faux** ou **indéfini**, le programme continue son exécution. L'assignation d'un tableau  $x[\mathbf{X}, n] := e$  est exécutée en évaluant  $e$   $n$  fois, dans un environnement  $\Omega$  étendu par la valeur de **X** correspondant à l'indice calculé. La séquence entre une commande et un programme est exécutée de manière usuelle.

### 3.5 Sûreté du typage

Le langage étant à la fois simple et permissif, nous arrivons à prouver que les programmes bien typés s'exécutent sans erreur (excepté les erreurs d'assertions). Pour définir la sûreté du typage, il faut d'abord s'assurer de la cohérence entre les environnements  $\Gamma$  et  $\Omega$  :

**Définition** ( $\Gamma \triangleright \Omega$ ). On dit que le contexte d'évaluation  $\Omega$  est correctement représenté par le contexte de typage  $\Gamma$ , et l'on note  $\Gamma \triangleright \Omega$  si :

- Les types scalaires sont représentés par des valeurs du bon type :

$$\forall x \in \mathcal{V}, \tau \in T, \Gamma(x) = \text{scalaire}[\tau] \implies \exists v \in \langle \text{valeur} \rangle, \Omega(x) = v \wedge \emptyset \vdash \text{Val } v : \tau$$

- Les types tableaux sont représentés par des tableaux ayant le bon type :

$$\begin{aligned} \forall x \in \mathcal{V}, \tau \in T, \Gamma(x) = \text{tableau}[\tau] &\implies \exists n \in \mathbb{N}, \exists (v_0, \dots, v_{n-1}) \in \langle \text{valeur} \rangle^n, \\ \Omega(x) = (v_0, \dots, v_{n-1}) &\wedge \forall i \in \llbracket 0; n-1 \rrbracket, \emptyset \vdash \text{Val } v_i : \tau \end{aligned}$$

- Les variables non définies dans  $\Gamma$  ne le sont pas dans  $\Omega$  non plus :

$$\forall x \in \mathcal{V}, x \notin \text{dom } \Gamma \implies x \notin \text{dom } \Omega$$

**Théorème** (Sûreté du typage des expressions). Si  $\Gamma \triangleright \Omega$  et  $\Gamma \vdash e : \tau$ , alors il existe une valeur  $v$  telle que  $\Gamma \vdash e \Downarrow v$  et  $\emptyset \vdash \text{Val } v : \tau$ .

Jugement : $\boxed{\Omega \vdash e \Downarrow v}$ (« sous $\Omega$ , $e$ s'évalue vers $v$ »)			
$\frac{\text{D-VALUE} \quad v \in \langle \text{valeur} \rangle}{\Omega \vdash v \Downarrow v}$	$\frac{\text{D-VAR} \quad \Omega(x) = v}{\Omega \vdash x \Downarrow v}$	$\frac{\text{D-VAR-UNDEF} \quad x \notin \text{dom } \Omega}{\Omega \vdash x \Downarrow \text{indéfini}}$	$\frac{\text{D-COND-TRUE} \quad \Omega \vdash e_1 \Downarrow \text{vrai} \quad \Omega \vdash e_2 \Downarrow v_2}{\Omega \vdash \text{si } e_1 \text{ alors } e_2 \text{ sinon } e_3 \Downarrow v_2}$
$\frac{\text{D-COND-FALSE} \quad \Omega \vdash e_1 \Downarrow \text{faux} \quad \Omega \vdash e_3 \Downarrow v_3}{\Omega \vdash \text{si } e_1 \text{ alors } e_2 \text{ sinon } e_3 \Downarrow v_3}$	$\frac{\text{D-COND-UNDEF} \quad \Omega \vdash e_1 \Downarrow \text{indéfini}}{\Omega \vdash \text{si } e_1 \text{ alors } e_2 \text{ sinon } e_3 \Downarrow \text{indéfini}}$		$\frac{\text{D-TAB-UNDEF} \quad x \notin \text{dom } \Omega}{\Omega \vdash x[e] \Downarrow \text{indéfini}}$
$\frac{\text{D-INDEX} \quad \Omega(x) = (v_0, \dots, v_{n-1}) \quad \Omega \vdash e \Downarrow r \quad r \geq 0 \quad r < n \quad r' = \text{truncate}_{\mathbb{F}}(r)}{\Omega \vdash x[e] \Downarrow v_{r'}}$			$\frac{\text{D-INDEX-UNDEF} \quad \Omega \vdash e \Downarrow \text{indéfini}}{\Omega \vdash x[e] \Downarrow \text{indéfini}}$
$\frac{\text{D-INDEX-NEG} \quad \Omega \vdash e \Downarrow r \quad r < 0}{\Omega \vdash x[e] \Downarrow 0}$	$\frac{\text{D-INDEX-OUTSIDE} \quad \Omega \vdash e \Downarrow r \quad r \geq n}{\Omega \vdash x[e] \Downarrow \text{indéfini}}$	$\frac{\text{D-FUNC} \quad \Omega \vdash e_1 \Downarrow v_1 \quad \dots \quad \Omega \vdash e_n \Downarrow v_n}{\Omega \vdash f(e_1, \dots, e_n) \Downarrow f(v_1, \dots, v_n)}$	
Jugement : $\boxed{\Omega_c \vdash c \Rightarrow \Omega'_c}$ et $\boxed{\Omega_c \vdash P \Rightarrow \Omega'_c}$ (« sous $\Omega_c$ , $P$ produit $\Omega'_c$ »)			
$\frac{\text{D-ASSIGN} \quad \Omega_c \neq \text{erreur} \quad \Omega_c \vdash e \Downarrow v}{\Omega_c \vdash x := e \Rightarrow \Omega_c[x \mapsto v]}$	$\frac{\text{D-ASSERT-OTHER} \quad \Omega_c \neq \text{erreur} \quad \Omega_c \vdash e \Downarrow v \quad v \in \{\text{faux}, \text{indéfini}\}}{\Omega_c \vdash \text{si } e \text{ alors } \langle \text{erreur} \rangle \Rightarrow \Omega_c}$		
$\frac{\text{D-ASSERT-TRUE} \quad \Omega_c \neq \text{erreur} \quad \Omega_c \vdash e \Downarrow \text{vrai}}{\Omega_c \vdash \text{si } e \text{ alors } \langle \text{erreur} \rangle \Rightarrow \text{erreur}}$	$\frac{\text{D-ERROR}}{\text{erreur} \vdash c \Rightarrow \text{erreur}}$	$\frac{\text{D-SEQ} \quad \Omega_{c,0} \vdash c \Rightarrow \Omega_{c,1} \quad \Omega_{c,1} \vdash P \Rightarrow \Omega_{c,2}}{\Omega_{c,0} \vdash c ; P \Rightarrow \Omega_{c,2}}$	
$\frac{\text{D-ASSIGN-TABLE} \quad \Omega_c \neq \text{erreur} \quad \Omega_c[\mathbf{x} \mapsto 0] \vdash e \Downarrow v_0 \quad \dots \quad \Omega_c[\mathbf{x} \mapsto n-1] \vdash e \Downarrow v_{n-1}}{\Omega_c \vdash x[\mathbf{x}, n] := e \Rightarrow \Omega_c[x \mapsto (v_0, \dots, v_{n-1})]}$			

FIGURE 5 – Évaluation des expressions, des commandes et des programmes

Afin de gérer les erreurs levées par les conditions de vérification, la relation  $\triangleright$  est étendue aux environnements d'exécution des commandes de la façon suivante :

$$\Gamma \triangleright_c \Omega_c \iff \Omega_c = \text{erreur} \vee \Gamma \triangleright \Omega_c$$

Le théorème de sûreté du typage des commandes stipule qu'une commande bien typée dans des contextes en relation (au sens de  $\triangleright_c$ ) sera correctement exécutée, et que les contextes en sortie seront aussi en relation.

**Théorème** (Sûreté du typage des commandes). Si  $\Gamma \vdash c \Rightarrow \Gamma'$  et  $\Gamma \triangleright_c \Omega_c$ , alors il existe  $\Omega'_c$  tel que  $\Omega_c \vdash c \Rightarrow \Omega'_c$  et  $\Gamma' \triangleright_c \Omega'_c$ .

### 3.6 Formalisation en Coq du langage $\mu\text{M}$

Afin de spécifier de manière précise la sémantique de  $\mu\text{M}$  et d'avoir une plus grande confiance en nos résultats, nous l'avons formalisée en Coq. Les jugements de typage sont définis comme des prédicats inductifs, tandis que la sémantique est définie de manière exécutable. Les théorèmes

$f_1 \odot f_2, \odot \in \{+, -\}$	indéfini	$f_2 \in \mathbb{F}$	$f_1 \odot f_2, \odot \in \{\times, \div\}$	indéfini	$f_2 \in \mathbb{F}, f_2 \neq 0$
indéfini	indéfini	$0 \odot f_2$	indéfini	indéfini	indéfini
$f_1 \in \mathbb{F}$	$f_1 \odot 0$	$f_1 \odot f_2$	$f_1$	indéfini	$f_1 \odot f_2$
$b_1 \langle \text{logicop} \rangle b_2$	indéfini	$b_2 \in \mathbb{B}$	$f_1 \langle \text{compop} \rangle f_2$	indéfini	$f_2 \in \mathbb{F}$
indéfini	indéfini	indéfini	indéfini	indéfini	indéfini
$b_1 \in \mathbb{B}$	indéfini	$b_1 \langle \text{logicop} \rangle b_2$	$f_1 \in \mathbb{F}$	indéfini	$f_1 \langle \text{compop} \rangle f_2$
$f_1 \times 0 = 0$			$f_1 \div 0 = \text{indéfini}$		
$\text{present}(v) = (v \neq \text{indéfini})$			$\text{abs}(x) \equiv \text{si } x \geq 0 \text{ alors } x \text{ sinon } -x$		
$\text{null}(\text{indéfini}) = \text{indéfini}$			$\text{min}(x, y) \equiv \text{si } x \geq y \text{ alors } y \text{ sinon } x$		
$\text{null}(v \neq \text{indéfini}) = (v \neq 0)$			$\text{max}(x, y) \equiv \text{si } x \geq y \text{ alors } x \text{ sinon } y$		
$\text{pos}(x) \equiv x > 0$			$\text{pos\_ou\_nul}(x) \equiv x \geq 0$		
$\text{arr}(\text{indéfini}) = 0$			$\text{inf}(\text{indéfini}) = 0$		
$\text{arr}(f \in \mathbb{F}) = \text{round}_{\mathbb{F}}(f)$			$\text{inf}(f \in \mathbb{F}) = \text{truncate}_{\mathbb{F}}(f)$		

FIGURE 6 – Comportement des fonctions

de sûreté énoncés précédemment ont été prouvés en Coq. Le développement fait actuellement 850 lignes, et est disponible publiquement<sup>5</sup>.

### 3.7 Compilateur du langage M et évaluation

**Implémentation** Nous avons implémenté en OCaml un compilateur appelé MLANG, capable de lire le code source mis à disposition par la DGFIP, d'ordonnancer ses assignations et d'interpréter le programme résultant, ou de le traduire vers du code Python exécutable. Ce compilateur est disponible publiquement<sup>6</sup> sous license GPL. Le compilateur permet également de spécialiser le code et de lui appliquer des optimisations classiques (évaluation partielle, numérotage global des valeurs, élimination de code mort).

**Évaluation** Nous avons ensuite pu vérifier que notre implémentation respectait la sémantique de M en utilisant des tests fournis par la DGFIP. Chaque test spécifie les valeurs de certaines variables d'entrées, ainsi que des valeurs de variable à vérifier ; une seule contrainte entre variable et valeur est définie par ligne. 542 tests nous ont été fournis ; les tests font de 74 à 713 lignes, avec une médiane à 597 lignes. Nous passons actuellement 104 tests, ayant chacun entre 558 et 674 lignes. L'examen des tests ne passant pas nous a permis de mettre à jour des informations manquantes quand à l'exécution du code M non encore publié par la DGFIP. Plus précisément, certaines valeurs d'entrées nécessaires aux tests ne sont pas spécifiées dans les tests mais par une autre application. La validation de notre artefact sur le jeu complet de tests de la DGFIP est donc laissé en travail à venir.

## 4 Étude formelle de l'impôt

Grâce à la sémantique du langage M, le code utilisé par la DGFIP pour calculer l'impôt sur le revenu devient une formalisation de la portion algorithmique du code des impôts. Il est

5. [https://gitlab.inria.fr/verifisc/mlang/tree/master/formal\\_semantics](https://gitlab.inria.fr/verifisc/mlang/tree/master/formal_semantics)

6. <https://gitlab.inria.fr/verifisc/mlang/tree/master/src/mlang>

possible d'utiliser des techniques classiques de méthodes formelles afin d'analyser le code. De plus, le fait que le langage M ne soit pas Turing-complet nous simplifie la tâche.

Afin d'explorer les applications potentielles de notre formalisation sans attendre la validation complète à venir de MLANG, nous avons décidé de créer un deuxième artefact, indépendant du code M, dans lequel nous réimplémentons un cas simplifié de l'impôt sur le revenu, auquel nous avons ajouté le calcul de diverses allocations. Les résultats de cette section sont donc basés sur ce deuxième artefact.

## 4.1 Utilisation de solveurs SMT

À condition de modéliser un montant d'argent comme un nombre entier de centimes (ou un réel à précision fixe), il est possible d'encoder le programme M de calcul des impôts dans un fragment logique contenant uniquement l'arithmétique et la logique du premier ordre. Il est alors possible d'encoder le calcul de l'impôt dans un solveur SMT comme Z3 [3].

Nous avons testé empiriquement plusieurs encodages SMT pour le calcul arithmétique de l'impôt : l'arithmétique entière non-bornée (avec un nombre entier de centimes), l'arithmétique réelle non-bornée (1,23 € étant encodé comme 1.23) et l'arithmétique entière bornée (encodée comme opérations sur des vecteurs de bits). Cet encodage a été testé sur l'impôt sur les revenus de 2017, qui ne relève pas du fragment linéaire de l'arithmétique. En effet, au moins une provision du codes des impôts de l'époque, la réduction d'impôt prévue au (b) du 4 du I de l'article 197 du CGI<sup>7</sup> dépend du carré du revenu fiscal de référence du foyer.

Dans ces conditions, seul l'encodage à base de vecteurs de bits nous a permis empiriquement d'obtenir une réponse de manière systématique sans tomber dans l'incomplétude des procédures de décision de Z3. Un encodage plus performant serait probablement possible, d'autant plus que la réforme du barème de l'impôt sur le revenu de 2019 supprime la seule partie non-linéaire du calcul. Cependant, nous considérons ici uniquement l'encodage sur vecteurs de bits.

Dans ce cas, quelle taille de vecteur choisir ? Étant donné qu'un vecteur de bits représente un nombre entier de centimes, il nous faut estimer la valeur intermédiaire maximale atteinte lors du calcul de l'impôt afin d'éviter tout débordement arithmétique. Dans le fragment linéaire du calcul, cette valeur maximale est atteinte lorsque l'on veut prendre une fraction d'un revenu. Plus la fraction que l'on veut prendre est précise, plus la valeur intermédiaire atteinte est grande. Pour calculer 42,8% d'un revenu  $X$ , il faut calculer  $X \times 428/1000$ . Dans notre code, nous nous limitons à une précision de l'ordre du dixième de pourcentage (donc un facteur multiplicatif de moins de 1000), suffisante pour approximer le calcul officiel à un ou deux euros près. La provision non-linéaire est encodée de manière spéciale sur un vecteur de bits deux fois plus grand pour éviter les débordements. Avec ces contraintes, si l'on veut pouvoir modéliser plus de 99% des cas avec un revenu annuel maximal de 105 000 euros par individu du foyer<sup>8</sup>, alors un nombre de bits égal à 34 suffit. On peut choisir un nombre de bits plus élevé pour calculer l'impôt de plus hauts revenus mais ceci impacte le temps de calcul de Z3.

Nous avons développé en utilisant l'interface Python de Z3 un prototype open-source<sup>9</sup> modélisant le calcul de l'impôt sur le revenu ainsi que diverses allocations pour les ménages. Ce prototype fait un certain nombre d'hypothèses qui restreignent l'espace possible des ménages considérés ; globalement, on considère des locataires n'ayant que des revenus salariaux. Cependant, toutes les combinaisons de situation matrimoniale (y compris le concubinage) et de distribution d'enfants à charge (là aussi le nombre maximal est borné, par défaut à 6) sont

7. <https://bofip.impots.gouv.fr/bofip/2495-PGP>

8. [https://www.insee.fr/fr/statistiques/fichier/3549487/REVPMEN18\\_D1\\_tres-hauts-revenus.pdf](https://www.insee.fr/fr/statistiques/fichier/3549487/REVPMEN18_D1_tres-hauts-revenus.pdf), p.4

9. <https://gitlab.inria.fr/verifisc/verifisc-python>

Caractéristique	Valeur avant	Valeur après	Variation
Salaire mensuel net premier individu	<b>2 760,76 €</b>	<b>3 010,76 €</b>	<b>+ 250,00 €</b>
Salaire mensuel net deuxième individu	<b>0,00 €</b>	0,00 €	0,00 €
Revenus annuels des salaires du foyer	33 129,12 €	36 129,12 €	<b>+ 3 000,00 €</b>
Revenu fiscal de référence (annuel)	29 816,00 €	32 516,00 €	<b>+ 2 700,00 €</b>
Impôt sur le revenu (annuel)	3 147,00 €	3 957,00 €	<b>+ 810,00 €</b>
Prime d'activité (mensuelle)	110,00 €	0,00 €	<b>- 110,00 €</b>
Allocations familiales (mensuelles)	132,00 €	132,00 €	0,00 €
Allocation rentrée scolaire (annuelle)	806,00 €	0,00 €	<b>- 806,00 €</b>
Bourse collège (trimestrielle)	0,00 €	0,00 €	0,00 €
Bourse lycée (annuelle)	0,00 €	0,00 €	0,00 €
Allocations Personnalisée au Logement (mensuelle)	0,00 €	0,00 €	0,00 €
Net touché après redistribution	33 692,12 €	33 756,12 €	<b>+ 64,00 €</b>
Augmentation des revenus mensuels après redistribution			<b>5,33 €</b>
Taux marginal			<b>97,9 %</b>

FIGURE 7 – Couple en concubinage avec deux enfants de 15 et 17 ans scolarisés et à charge du deuxième individu (sans activité). Le ménage habite en zone II dans une location pour 897,75 €. On suppose que le premier individu du couple est augmenté de 250 € par mois. L'augmentation de l'impôt sur le revenu, ainsi que la perte de la prime d'activité et de l'allocation de rentrée scolaire, conduisent à ce que le couple touche uniquement 2,1 % de l'augmentation initiale.

prises en compte par le modèle. Z3 nous permet alors de répondre à des questions de la forme « Existe-t-il un ménage  $X$  tel que  $P(X)$  », où  $P(X)$  est un prédicat numérique dépendant des caractéristiques du ménage  $X$  et du montant des différents impôts et revenus calculés. Il est possible d'encoder dans  $P$  des questions impliquant l'évolution du ménage entre deux années, soit  $P(X) = Q(X, f(X))$  où  $f$  décrit la transformation du ménage d'une année sur l'autre (en supposant pour notre prototype que la législation fiscale ne change pas) et  $Q$  étant notre prédicat numérique.

Aussi, en prenant pour  $f$  l'augmentation du salaire du premier individu du ménage de  $x$  euros par mois, et pour  $Q$  le fait que l'augmentation du revenu après redistribution (impôts et allocations) du ménage représente moins de  $y$  % des  $x$  euros d'augmentation, alors Z3 nous permet de répondre à la question de l'existence d'une taxation marginale plus élevée que  $(100 - y)$  %. En effet, le taux marginal de taxation est défini comme le taux d'impôt prélevé sur une petite augmentation de revenus par rapport à une situation initiale. Plus précisément, nous considérons le taux marginal effectif de prélèvement (TMEP) car nous prenons en compte à la fois impôts et allocations.

La figure 7 présente une des réponses trouvées par Z3 à la recherche de taux marginaux élevés. La correction de l'encodage du calcul de l'impôt dans le prototype a été testée sur de nombreux exemples en comparant avec les simulateurs officiels de l'impôt sur le revenu et des aides sociales. Les montants sont corrects à 1-2 € près, la différence étant sûrement imputable à des divergences concernant l'arrondi des montants dans le calcul et aux approximations de fractions liées à l'encodage (décrites plus haut). Par contre, la correction des résultats négatifs retournés par Z3 (« il n'existe pas de foyer tel que ... ») est conditionnée à l'absence de bugs.

La question de la recherche de taux marginaux élevés est une question complexe, qui nécessite « d'inverser » la fonction de calcul de l'impôt, puisque l'on cherche des foyers fiscaux à partir d'un montant d'impôt payé. Cette inversion réalisée par le solveur SMT est une amélioration inédite par rapport aux autres outils d'analyse utilisés en économie pour évaluer les effets d'une

formule de calcul de l'impôt. En effet, les travaux récents évaluant ce TMEP [5] utilisent une dérivation partielle manuelle qui néglige les arrondis à l'euro près et une augmentation de revenus fixe pour tous les ménages à 3 % des revenus professionnels. Le calcul se fait sur un jeu de données correspondant à des ménages réels, mais qui couvre généralement les revenus sur une année et ne permettent pas de suivre l'évolution d'un ménage. De plus, l'accès à ces données est protégé par le secret fiscal. Pour cette raison, traditionnellement, seul l'Insee peut produire de telles analyses.

La possibilité d'explorer tout l'espace des ménages possibles permet également d'anticiper des situations indésirables résultant par exemple de l'interaction entre plusieurs dispositifs législatifs, comme l'illustre la figure 7. L'analyse économique des transferts sociaux aidée par SMT ouvre donc des pistes prometteuses qui dépassent le cadre de cet article. Cependant, la question du passage à l'échelle va se révéler cruciale : en effet, notre prototype prend déjà plusieurs heures et des dizaines de Go de mémoire vive afin de répondre aux requêtes portant sur l'impôt sur le revenu dans un cas simplifié. Nous avons donc besoin d'outils fiables pour guider le travail du solveur SMT et lui présenter des requêtes les plus simples possibles.

## 4.2 Travaux futurs : interprétation abstraite

Revenons maintenant au langage M et à la base de code existante maintenue par la DGFIP qui encode le calcul de l'impôt. Il est possible de traduire un programme M vers un encodage compatible avec un solveur SMT. Cependant, comme mentionné auparavant, cela produirait des requêtes trop volumineuses pour être traitées en temps raisonnable. Notre stratégie est donc de circonscrire le problème à une situation simplifiée par rapport au cas général : par exemple, le cas où le ménage est un couple marié qui ne possède que des revenus salariaux et des enfants sans aucun cas spécial. Dans ce cas simplifié, les optimisations classiques (élimination de code mort, numérotage global des valeurs, évaluation partielle), implémentées dans le compilateur MLANG, font passer le nombre de variables du programme d'environ 46 000 à 650. Mais même ainsi, le programme résultant est trop volumineux pour le solveur SMT. En effet, le code M contient des motifs récurrents comme celui de l'écrêtage de valeur par un seuil constant (expressions de la forme  $\text{VAR1} = \min(\text{VAR0}, 20)$ ). Pour simplifier ces motifs, une interprétation abstraite utilisant un domaine numérique semble être tout indiquée. Nous envisageons donc dans le futur de connecter MLANG à l'outil d'interprétation abstraite MOPSA [7], et d'utiliser les informations fournies par l'analyse afin de simplifier le programme sous un ensemble d'hypothèses sur les données en entrée (revenus positifs, etc).

Nous espérons que l'interprétation abstraite, combinée à une division des requêtes en sous-problèmes spécialisés sur critères décidés par l'utilisateur (par exemple le cas célibataire, le cas marié et le cas concubinage), nous permette de produire des requêtes optimisées traitables rapidement par un solveur SMT avec un encodage des entiers en vecteurs de bits.

## 5 Travaux connexes

Diverses initiatives ont déjà été tentées afin de formaliser une petite partie d'une législation ou d'une réglementation. Certaines sont très anciennes comme celle de Allen [1] en 1956, qui affine sa méthode en 1978 [2] pour utiliser un langage de diagrammes. Kowalski et al. [14] utilisent Prolog en 1986 sur la détermination de la nationalité britannique. Plus récemment, Sarah Lawsky a mené deux études de cas [8, 9] portant sur des fragments du droit fiscal américain, et propose d'utiliser la logique par défaut [13] pour encoder la structure de la législation. La difficulté que rencontrent toutes ces initiatives provient de la traduction de la loi en algorithme :

seuls certains fragments de la législation s'y prêtent, et même dans ce cas le texte de loi n'est pas écrit dans un style programmatique, ce qui rend la traduction ardue.

En ce qui concerne les contrats, Hvitved propose une spécification de contrats multipartites qu'il formalise [6] dans un DSL. Il existe un très grand nombre de travaux portant sur ce sujet, originant des communautés de la logique formelle, du droit, de l'intelligence artificielle (avant l'apparition de l'apprentissage statistique), et plus récemment des systèmes distribués (blockchain et « smart contracts »).

Ce domaine de recherche est également exploré par des sociétés privées. En Espagne, une initiative menée par la société Formal Vindications a formalisé les conséquences d'une réglementation européenne sur la conduite [4] pour mieux en montrer les incohérences. Au Royaume-Uni, la société Aesthetic Integration formalise le fonctionnement des plate-formes des marchés financiers et prouve leur conformité par rapport aux réglementations financières [11]. Ces travaux constituent sans doute l'initiative la plus aboutie de formalisation de texte législatif, qui se base sur un langage et une suite logicielle appelée Imandra combinant plusieurs techniques issues de la recherche en méthodes formelles.

Parallèlement à ces efforts de formalisation, un autre axe de recherche est l'implémentation de textes législatifs dans une base de données de règles couplée à un moteur d'exécution. En Nouvelle-Zélande, l'outil Datalex [10] est utilisé pour offrir une interface de type « chatbot » à des utilisateurs qui désireraient comparer une situation à la législation existante. Cette approche est à rapprocher de la catégorie des « Business Rules Management Systems » comme Drools [12], qui n'a pas pour objectif une formalisation complète mais plutôt le développement d'un outil pratique. On peut également citer l'initiative OpenFisca<sup>10</sup> menée par la Direction Interministérielle du Numérique et des Systèmes d'Information et de Communication, qui propose une implémentation en Python du système socio-fiscal français.

À notre connaissance, notre travail est inédit quant à l'approche systématique et la taille de la portion de législation formalisée. En effet, la France est le seul pays ayant publié en open-source le code calculant l'impôt sur le revenu de ses concitoyens. La plupart des travaux existants se focalisent sur des contrats, plus limités dans leur portée. La formalisation de la législation existante est moins intéressante à cause de la taille imposante des textes de lois et de leur inadaptation au raisonnement logique (incohérences, imprécisions). L'implémentation en M du code des impôts, maintenue depuis trois décennies par la DGFIP au prix d'un travail important, a été la brique de base indispensable à notre étude.

## 6 Conclusion

L'application des méthodes formelles à l'artefact logiciel publié par la DGFIP ouvre des pistes très prometteuses quant à l'analyse de l'impact de l'impôt sur le revenu. Le langage M, bien qu'ancien, relève d'un choix éclairé d'architecture logicielle dans le contexte de l'administration publique des années 90. La centralisation de l'encodage du code des impôts à l'aide d'un DSL dans une implémentation unique et mutualisée entre les applications clientes nous a permis de concentrer efficacement notre travail de formalisation. La formalisation de M présentée dans cet article, permet également la compilation de M vers virtuellement n'importe quel autre langage, de manière à fournir une implémentation clés en main du code des impôts à toutes les applications qui en auraient besoin : analyses macroéconomiques, simulateurs en ligne, etc.

Néanmoins, la correction du code M par rapport au code des impôts reste un sujet critique. En effet, un bug d'implémentation est potentiellement source de contentieux entre l'État et le

---

10. <https://openfisca.org>



particulier lésé par un calcul de l'impôt erroné. Le langage M n'a pas une structure permettant de se convaincre ou de vérifier modulairement cette correction fonctionnelle. La prochaine étape de ce travail est donc de créer un langage inspiré de M, mais qui serait utilisé pour annoter la loi dans un style de programmation littérale, article par article. Ces annotations tiendraient lieu de spécification formelle et pourraient aussi guider le législateur, à l'aide d'un outillage adapté mettant en valeur grâce à de l'inférence des incohérences ou des cas non-spécifiés.

## Références

- [1] Layman E Allen. Symbolic logic : A razor-edged tool for drafting and interpreting legal documents. *Yale LJ*, 66 :833, 1956.
- [2] Layman E. Allen and C. Rudy Engholm. Normalized legal drafting and the query method. *Journal of Legal Education*, 29(4) :380–412, 1978.
- [3] Leonardo de Moura and Nikolaj Bjørner. Z3 : An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
- [4] D Fernández Duque, M González Bedmar, D Sousa, Joosten, J.J, and G. Errezil Alberdi. To drive or not to drive : A formal analysis of requirements (51) and (52) from regulation (eu) 2016/799. In *Personalidades jurídicas difusas y artificiales*, pages 159–171. TransJus Working Papers Publication - Edición Especial, 4 2019.
- [5] Juliette Fourcot and Michaël Sicsic. Les taux marginaux effectifs de prélèvement pour les personnes en emploi en france en 2014, 2017.
- [6] Tom Hvitved. *Contract formalisation and modular implementation of domain-specific languages*. PhD thesis, Citeseer, 2011.
- [7] M. Journault, A. Miné, M. Monat, and A. Ouadjaout. Combinations of reusable abstract domains for a multilingual static analyser. In *Proc. of VSTTE19*, pages 1–17, Jul. 2019.
- [8] Sarah B. Lawsky. Formalizing the Code. *Tax Law Review*, 70(377), 2017.
- [9] Sarah B. Lawsky. A Logic for Statutes. *Florida Tax Review*, 2018.
- [10] Andrew Mowbray, Philip Chung, and Graham Greenleaf. Utilising ai in the legal assistance sector – testing a role for legal information institutes. In *LegalAIIA*, 2019.
- [11] Grant Olney Passmore and Denis Ignatovich. Formal verification of financial algorithms. In *CADE*, 2017.
- [12] Mark Proctor. Drools : A rule engine for complex event processing. In *Proceedings of the 4th International Conference on Applications of Graph Transformations with Industrial Relevance*, AGTIVE'11, pages 2–2, Berlin, Heidelberg, 2012. Springer-Verlag.
- [13] R. Reiter. Readings in nonmonotonic reasoning. chapter A Logic for Default Reasoning, pages 68–93. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987.
- [14] M. J. Sergot, F. Sadri, R. A. Kowalski, F. Kriwaczek, P. Hammond, and H. T. Cory. The british nationality act as a logic program. *Commun. ACM*, 29(5) :370–386, May 1986.
- [15] R. Tarjan. Depth-first search and linear graph algorithms. In *12th Annual Symposium on Switching and Automata Theory (swat 1971)*, pages 114–121, Oct 1971.