HAL
open science

A Modern Compiler for the French Tax Code

Denis Merigoux, Raphaël Monat, Jonathan Protzenko

To cite this version:

Denis Merigoux, Raphaël Monat, Jonathan Protzenko. A Modern Compiler for the French Tax Code. CC '21: 30th ACM SIGPLAN International Conference on Compiler Construction, Mar 2021, Virtual, South Korea. pp.71-82, 10.1145/3446804.3446850 . hal-03002266v3

HAL Id: hal-03002266

https://inria.hal.science/hal-03002266v3

Submitted on 25 Jan 2021

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.

A Modern Compiler for the French Tax Code

Denis Merigoux*
Inria
Paris, France
denis.merigoux@inria.fr

Raphaël Monat*
Sorbonne Université, CNRS, LIP6
Paris, France
raphael.monat@lip6.fr

Jonathan Protzenko
Microsoft Research
USA
protz@microsoft.com

Abstract

In France, income tax is computed from taxpayers' individual returns, using an algorithm that is authored, designed and maintained by the French Public Finances Directorate (DGFiP). This algorithm relies on a legacy custom language and compiler originally designed in 1990, which unlike French wine, did not age well with time. Owing to the shortcomings of the input language and the technical limitations of the compiler, the algorithm is proving harder and harder to maintain, relying on ad-hoc behaviors and workarounds to implement the most recent changes in tax law. Competence loss and aging code also mean that the system does not benefit from any modern compiler techniques that would increase confidence in the implementation.

We overhaul this infrastructure and present Mlang, an open-source compiler toolchain whose goal is to replace the existing infrastructure. Mlang is based on a reverse-engineered formalization of the DGFiP's system, and has been thoroughly validated against the private DGFiP test suite. As such, Mlang has a formal semantics; eliminates previous hand-written workarounds in C; compiles to modern languages (Python); and enables a variety of instrumentations, providing deep insights about the essence of French income tax computation. The DGFiP is now officially transitioning to Mlang for their production system.

Keywords: legal expert system, compiler, tax code

1 Introduction

The French Tax Code is a body of legislation amounting to roughly 3,500 pages of text, defining the modalities of tax collection by the state. In particular, each new fiscal year, a new edition of the Tax Code describes in natural language how to compute the final amount of income tax (IR, for *impôt sur le revenu*) owed by each household.

As in many other tax systems around the world, this computation is quite complex. France uses a bracket system (as in, say, the US federal income tax), along with a myriad of tax credits, deductions, optional rules, state-sponsored direct aid, all of which are parameterized over the composition of the household, that is, the number of children, their respective ages, potential disabilities, and so on.

*Equal contribution

Preprint, Jan. 2021, Online
2021.

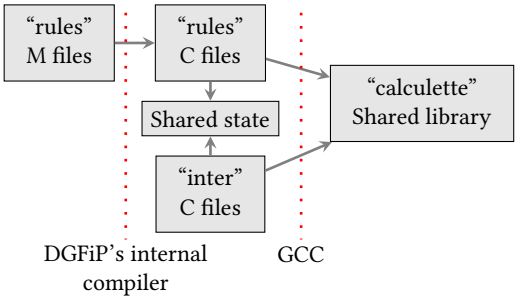

Figure 1. Legacy architecture

Unlike, say, the United States, the French system relies heavily on automation. During tax season, French taxpayers log in to the online tax portal, which is managed by the state. There, taxpayers are presented with online forms, generally pre-filled. If applicable, taxpayers can adjust the forms, e.g. by entering extra deductions or credits. Once the taxpayer is satisfied with the contents of the online form, they send in their return. Behind the scenes, the IR algorithm is run, and taking as input the contents of the forms, returns the final amount of tax owed. The taxpayer is then presented with the result at tax-collection time.

Naturally, the ability to independently reproduce and thus trust the IR computation performed by the DGFiP is crucial. First, taxpayers need to *understand* the result, as their own estimate may differ (explainability). Second, citizens may want to *audit* the algorithm, to ensure it faithfully implements the law (correctness). Third, a standalone, reusable implementation allows for a complete and precise *simulation* of the impacts of a tax reform, greatly improving existing efforts [8, 17] (forecasting).

Unfortunately, we are currently far from a transparent, open-source, reproducible computation. Following numerous requests (using a disposition similar to the United States' Freedom of Information Act), parts of the existing source code were published. In doing so, the public learned that *i)* the existing infrastructure is made up of various parts pieced together and that *ii)* key data required to accurately reproduce IR computations was not shared with the public.

The current, legacy architecture of the IR tax system is presented in Fig. 1. The bulk of the tax code is described as a set of "rules" authored in M, a custom, non Turing-complete language. A total of 90,000 lines of M rules compile to 535,000 lines of C (including whitespace and comments) via a custom

compiler. Rules are now mostly public [10]. Over time, the expressive power of rules turned out to be too limited to express a particular feature, known as *liquidations multiples*, which involves tax returns across different years. Lacking the expertise to extend the M language, the DGFiP added in 1995 some high-level glue code in C, known as "inter". The glue code is closer to a full-fledged language, and has a non-recursive call-graph which may call the "rules" computation multiple times with various parameters. The "inter" driver amounts to 35,000 lines of C code and has not been released.

Both "inter" and "rules" are updated every year to follow updates in the law, and as such, have been extensively modified over their 30-year operational lifespan.

Our goal is to address these shortcomings by bringing the French tax code infrastructure into the 21st century. Specifically, we wish to: *i)* reverse-engineer the unpublished parts of the DGFiP computation, so as to *ii)* provide an explainable, open-source, *correct* implementation that can be independently audited; furthermore, we wish to *iii)* modernize the compiler infrastructure, eliminating in the process any hand-written C that could not be released because of security concerns, thus enabling a host of modern applications, simulations and use-cases.

- We start with a reverse-engineered formal semantics for the M DSL, along with a proof of type safety performed using the Coq [31] proof assistant (Section 2).
- To eliminate C code from the ecosystem, we extend the M language with enough capabilities to encode the logic of the high-level "inter" driver (Fig. 1) – we dub the new design M++ (Section 3).
- To execute M/M++ programs, we introduce Mlang, a complete re-implementation which combines a reference interpreter along with an optimizing compiler that generates C and Python code (Section 4).
- We evaluate our implementation: we show how we attained 100% conformance on the legacy system's testsuite, then proceed to enable a variety of analyses and instrumentations to fuzz, measure and stress-test our new system (Section 5).
- We conclude with a *tour d'horizon* of related attempts at increasing trust in algorithmic parts of the law (Section 6).

Our code is open-source and available on GitHub [20] and as an archived artifact on Zenodo [21]. We have engaged with the DGFiP, and following numerous discussions, iterations, and visits to their offices, we have been formally approved to start replacing the legacy infrastructure with our new implementation, meaning that within a few years' time, all French tax returns will be processed using the compiler described in the present paper.

2 Giving Semantics to the M Language

The 2018 version of the income tax computation [10] is split across 48 files, for a total of 92,000 lines of code. The code is written in M, the input language originally designed by the DGFiP. In order to understand this body of tax code, we set out to give a semantics to M.

2.1 Overview of M

M programs are made up of two parts: declarations and rules.

Declarations introduce: input variables, intermediary variables, output variables and exceptions. Variables are either scalars or fixed-length arrays. Both variables and exceptions are annotated with a human-readable description. Variables that belong to the same section of the tax form are annotated with the same kind. Examples of kinds include "triggers tax credit", or "is advance payment". This is used later in M++ (Section 3.3) for partitioning variables, and quickly checking whether any variable of a given kind has a non-`undef` value.

Rules capture the computational part of an M program; they are either variable assignments or raise-if statements.

As a first simplified example, the French tax code declares an input variable `V_0AC` for whether an individual is single (value `1`) or not (value `0`). Lacking any notion of data type or enumeration, there is no way to enforce statically that an individual cannot be married (`V_0AM`) and single (`V_0AC`) at the same time. Instead, an exception `A031` is declared, along with a human-readable description. Then, a rule raises an exception if the sum of the two variables is greater than 1. (The seemingly superfluous `+ 0` is explained in Section 2.5.) For the sake of example, we drop irrelevant extra syntactic features, and for the sake of readability, we translate keywords and descriptions into English.

```
V_0AC : input family ... : "Checkbox : Single" type BOOLEAN ;
V_0AM : input family ... : "Checkbox : Married" type BOOLEAN ;
A031:exception :"A":"031":"00":"both married and single":"N";

if V_0AC + V_0AM + 0 > 1 then error A031 ;
```

As a second simplified example, the following M rule computes the value of a hypothetical variable `TAXBREAK`. Its value is computed from variables `CHILDRENCOUNT` (for the number of children in the household) and `TAXOWED` (for the tax owed before the break) – the assigned expression relies on a conditional and the built-in `max` function. This expression gives a better tax break to households having three or more children.

```
TAXBREAK= if (CHILDRENCOUNT+0 > 2)
  then max(MINTAXBREAK,TAXOWED * 20 / 100)
  else MINTAXBREAK endif;
```

For the rest of this paper, we abandon concrete syntax and all-caps variable names, in favor of a core calculus that faithfully models M: μM.

2.2 μM: a core model of M

The μM core language omits variable declarations, whose main purpose is to provide a human-readable description

string that relates them to the original tax form. The μM core language also eliminates syntactic sugar, such as statically bounded loops, or type aliases (e.g. `BOOLEAN`). Finally, a particular feature of M is that rules may be provided in any order: the M language has a built-in dependency resolution feature that automatically re-orders computations (rules) and asserts that there are no loops in variable assignments. In our own implementation (Mlang, Section 4), we perform a topological sort; in our μM formalization, we assume that computations are already in a suitable order.

2.3 Syntax of μM

We describe the syntax of μM in Fig. 2. A program is a series of statements ("rules"). Statements are either raise-error-if, or assignments. We define two forms of assignment: one for scalars and the other for fixed-size arrays. The latter is of the form `a[X, n] := e`, where `X` is bound in `e` (the index is *always* named `X`). Using Haskell's list comprehension syntax, this is to be understood as $a := [e|X \leftarrow [0..n-1]]$.

Expressions are a combination of variables (including the special index expression `X`), values, comparisons, logic and arithmetic expressions, conditionals, calls to builtin functions, or index accesses. Most functions exhibit standard behavior on floating-point values, but M assumes the default IEEE-754 rounding mode, that is, rounding to nearest and ties to even. The detailed behavior of each function is described in Fig. 6.

Values can be `undef`, which arises in two situations: references to variables that have not been defined (i.e. for which the entry in the tax form has been left blank) and out of bounds array accesses. All other values are IEEE-754 double-precision numbers, i.e. 64-bit floats. The earlier `BOOLEAN` type (Section 2.1) is simply an alias for a float whose value is implicitly `0` or `1`. There is no other kind of value, as a reference to an array variable is invalid. Function `present` discriminates the `undef` value from floats.

2.4 Typing μM

Types in μM are either scalar or array types. M does not offer nested arrays. Therefore, typing is mostly about making sure scalars and arrays are not mixed up.

In Fig. 3, a first judgment $\boxed{\Gamma \vdash e}$ defines expression well-formedness. It rules out references to arrays, hence enforcing that expressions have type scalar and that no values of type array can be produced. Furthermore, variables may have no assignment at all (if the underlying entry in the tax form has been left blank) but may still be referred in other rules. Rather than introduce spurious variable assignments with `undef`, we remain faithful to the very loose nature of the M language and account for references to undefined variables.

Then, $\boxed{\Gamma \vdash \langle program \rangle \Rrightarrow \Gamma'}$ enforces well-formedness for a whole program while returning an extended environment

$\langle program \rangle$ ::= $\langle command \rangle$ | $\langle command \rangle$; $\langle program \rangle$

$\langle command \rangle$ ::= `if` $\langle expr \rangle$ `then` $\langle error \rangle$
| $\langle var \rangle$ `:=` $\langle expr \rangle$ | $\langle var \rangle$ `[ X ;` $\langle float \rangle$ `] :=` $\langle expr \rangle$

$\langle expr \rangle$ ::= $\langle var \rangle$ | `X` | $\langle value \rangle$ | $\langle expr \rangle$ $\langle binop \rangle$ $\langle expr \rangle$
| $\langle unop \rangle$ $\langle expr \rangle$ |`if` $\langle expr \rangle$ `then` $\langle expr \rangle$ `else` $\langle expr \rangle$
| $\langle func \rangle$ `(` $\langle expr \rangle, \ldots, \langle expr \rangle$ `)` | $\langle var \rangle$ `[` $\langle expr \rangle$ `]`

$\langle value \rangle$ ::= `undef` | $\langle float \rangle$

$\langle binop \rangle$ ::= $\langle arithop \rangle$ | $\langle boolop \rangle$

$\langle arithop \rangle$ ::= `+` | `-` | `*` | `/`

$\langle boolop \rangle$ ::= `<=` | `<` | `>` | `>=` | `==` | `!=` | `&&` | `||`

$\langle unop \rangle$ ::= `-` | `~`

$\langle func \rangle$ ::= `round` | `truncate` | `max` | `min` | `abs`
| `pos` | `pos_or_null` | `null` | `present`

Figure 2. Syntax of the μM language

Γ'. We take advantage of the fact that scalar and array assignments have different syntactic forms. M disallows assigning different types to the same variable; we rule this out in T-Assign-*. A complete μM program is well-formed if $\varnothing \vdash P \Rrightarrow _$.

2.5 Operational semantics of μM

At this stage, seeing that there are neither unbounded loops nor user-defined (recursive) functions in the language, M is obviously *not* Turing-complete. The language semantics are nonetheless quite devious, owing to the `undef` value, which can be explicitly converted to a float via `a + 0`, as seen in earlier examples. We proceed to formalize them in Coq [31], using the Flocq library [4]. This ensures we correctly account for all cases related to the `undef` value, and guides the implementation of Mlang (Section 3).

Expressions. The semantics of expressions is defined in Fig. 4. The memory environment, written Ω is a function from variables to either scalar values (usually denoted v), or arrays (written $(v_0, \ldots, v_{n-1})$). A value absent from the environment evaluates to `undef`.

The special array index variable `X` is evaluated as a normal variable. Conditionals reduce normally, except when the guard is `undef`: in that case, the whole conditional evaluates into `undef`. If an index evaluates to `undef`, the whole array access is `undef`. In the case of a negative out-of-bounds index access the result is 0; in the case of a positive out-of-bounds index access the result is `undef`. Otherwise, the index is truncated into an integer, used to access Ω. The behavior of functions, unary and binary operators is described in Fig. 6.

Figuring out these (unusual) semantics took over a year. We initially worked in a black-box setting, using as an oracle for our semantics the simplified online tax simulator offered by the DGFiP. After the initial set of M rules was open-sourced, we simply manually crafted test cases and fed those

Global function environment Δ:

$\Delta(\texttt{round}) = \Delta(\texttt{truncate}) = \Delta(\texttt{abs}) = \Delta(\texttt{pos})$
$= \Delta(\texttt{pos_or_null}) = \Delta(\texttt{null}) = \Delta(\texttt{present}) = 1$
$\Delta(\texttt{min}) = \Delta(\texttt{max}) = \Delta(\langle \mathit{arithop} \rangle) = \Delta(\langle \mathit{boolop} \rangle) = 2$

Judgment : $\boxed{\Gamma \vdash e}$ ("Under Γ, e is well-formed")

$$\textsc{T-float}\ \frac{}{\Gamma \vdash \langle \mathit{float} \rangle} \qquad \textsc{T-undef}\ \frac{}{\Gamma \vdash \texttt{undef}} \qquad \textsc{T-var-undef}\ \frac{x \notin \mathrm{dom}\,\Gamma}{\Gamma \vdash x} \qquad \textsc{T-var}\ \frac{\Gamma(x) = \texttt{scalar}}{\Gamma \vdash x}$$

$$\textsc{T-index-undef}\ \frac{x \notin \mathrm{dom}\,\Gamma \qquad \Gamma \vdash e}{\Gamma \vdash x[e]} \qquad \textsc{T-conditional}\ \frac{\Gamma \vdash e_1 \qquad \Gamma \vdash e_2 \qquad \Gamma \vdash e_3}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3}$$

$$\textsc{T-index}\ \frac{\Gamma(x) = \texttt{array} \qquad \Gamma \vdash e}{\Gamma \vdash x[e]} \qquad \textsc{T-func}\ \frac{\Delta(f) = n \qquad \Gamma \vdash e_1 \qquad \cdots \qquad \Gamma \vdash e_n}{\Gamma \vdash f(e_1, \ldots, e_n)}$$

Judgment : $\boxed{\Gamma \vdash \langle \mathit{command} \rangle \Rrightarrow \Gamma'}$ and

$\boxed{\Gamma \vdash \langle \mathit{program} \rangle \Rrightarrow \Gamma'}$ ("P transforms Γ to Γ'")

$$\textsc{T-cond}\ \frac{\Gamma \vdash e}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ \langle \mathit{error} \rangle \ \Rrightarrow \Gamma} \qquad \textsc{T-seq}\ \frac{\Gamma_0 \vdash c \Rrightarrow \Gamma_1 \qquad \Gamma_1 \vdash P \Rrightarrow \Gamma_2}{\Gamma_0 \vdash c\ ;\ P \Rrightarrow \Gamma_2}$$

$$\textsc{T-assign-scalar}\ \frac{x \in \Gamma \Rightarrow \Gamma(x) = \texttt{scalar} \qquad \Gamma \vdash e}{\Gamma \vdash x := e \ \Rrightarrow \Gamma[x \mapsto \texttt{scalar}]}$$

$$\textsc{T-assign-array}\ \frac{x \in \Gamma \Rightarrow \Gamma(x) = \texttt{array} \qquad \Gamma[\texttt{X} \mapsto \texttt{scalar}] \vdash e}{\Gamma \vdash x[\texttt{X}, n] := e \ \Rrightarrow \Gamma[x \mapsto \texttt{array}]}$$

Figure 3. Typing of expressions and programs

by hand to the online simulator to adjust our semantics. This allowed us to gain credibility and to have the DGFiP take us seriously. After that, we were allowed to enter the DGFiP offices and browse the source of their M compiler, as long as we did not exfiltrate any information. This final "code browsing" allowed us to understand the "inter" part of their compiler, a well as nail down the custom operators from Fig. 15.

Statements. The memory environment Ω is extended into Ω_c, to propagate the error case that may be raised by exceptions. An assignment updates a valid memory environment with the computed value. If an assertion's guard evaluates to a non-zero float, an error is raised; otherwise, program execution continues. Rule D-error propagates a raised error across a program. The whole-array assignment works by evaluating the expression in different memory environments, one for each index.

2.6 Type safety

We now prove type safety in Coq. Owing to the unusual semantics of the `undef` value, and to the lax treatment of

Judgment : $\boxed{\Omega \vdash e \Downarrow v}$ ("Under Ω, e evaluates to v")

$$\textsc{D-value}\ \frac{v \in \langle \mathit{value} \rangle}{\Omega \vdash v \Downarrow v} \qquad \textsc{D-var-undef}\ \frac{x \notin \mathrm{dom}\,\Omega}{\Omega \vdash x \Downarrow \texttt{undef}} \qquad \textsc{D-var}\ \frac{\Omega(x) = v}{\Omega \vdash x \Downarrow v}$$

$$\textsc{D-cond-true}\ \frac{\Omega \vdash e_1 \Downarrow f \qquad f \notin \{0, \texttt{undef}\} \qquad \Omega \vdash e_2 \Downarrow v_2}{\Omega \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow v_2} \qquad \textsc{D-X}\ \frac{\Omega(\texttt{X}) = v}{\Omega \vdash \texttt{X} \Downarrow v}$$

$$\textsc{D-cond-false}\ \frac{\Omega \vdash e_1 \Downarrow 0 \qquad \Omega \vdash e_3 \Downarrow v_3}{\Omega \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow v_3} \qquad \textsc{D-index-neg}\ \frac{\Omega \vdash e \Downarrow r \qquad r < 0}{\Omega \vdash x[e] \Downarrow 0}$$

$$\textsc{D-cond-undef}\ \frac{\Omega \vdash e_1 \Downarrow \texttt{undef}}{\Omega \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow \texttt{undef}} \qquad \textsc{D-index-undef}\ \frac{\Omega \vdash e \Downarrow \texttt{undef}}{\Omega \vdash x[e] \Downarrow \texttt{undef}}$$

$$\textsc{D-index-outside}\ \frac{\Omega \vdash e \Downarrow r \qquad r \geqslant n \qquad |\Omega(x)| = n}{\Omega \vdash x[e] \Downarrow \texttt{undef}} \qquad \textsc{D-tab-undef}\ \frac{x \notin \mathrm{dom}\,\Omega}{\Omega \vdash x[e] \Downarrow \texttt{undef}}$$

$$\textsc{D-index}\ \frac{\Omega(x) = (v_0, \ldots, v_{n-1}) \qquad \Omega \vdash e \Downarrow r \qquad r \in [0, n) \qquad r' = \mathrm{truncate}_{\mathbb{F}}(r)}{\Omega \vdash x[e] \Downarrow v_{r'}}$$

$$\textsc{D-func}\ \frac{\Omega \vdash e_1 \Downarrow v_1 \qquad \cdots \qquad \Omega \vdash e_n \Downarrow v_n}{\Omega \vdash f(e_1, \ldots, e_n) \Downarrow f(v_1, \ldots, v_n)}$$

Figure 4. Operational semantics: expressions

Judgment : $\boxed{\Omega_c \vdash c \Rrightarrow \Omega_c'}$ and

$\boxed{\Omega_c \vdash P \Rrightarrow \Omega_c'}$ ("Under Ω_c, P produces Ω_c'")

$$\textsc{D-assign}\ \frac{\Omega_c \neq \texttt{error} \qquad \Omega_c \vdash e \Downarrow v}{\Omega_c \vdash x := e \Rrightarrow \Omega_c[x \mapsto v]}$$

$$\textsc{D-assert-other}\ \frac{\Omega_c \neq \texttt{error} \qquad \Omega_c \vdash e \Downarrow v \qquad v \in \{0, \texttt{undef}\}}{\Omega_c \vdash \texttt{if}\ e\ \texttt{then}\ \langle \mathit{error} \rangle \Rrightarrow \Omega_c}$$

$$\textsc{D-assert-true}\ \frac{\Omega_c \neq \texttt{error} \qquad \Omega_c \vdash e \Downarrow f \qquad f \notin \{0, \texttt{undef}\}}{\Omega_c \vdash \texttt{if}\ e\ \texttt{then}\ \langle \mathit{error} \rangle \Rrightarrow \texttt{error}}$$

$$\textsc{D-error}\ \frac{}{\texttt{error} \vdash c \Rrightarrow \texttt{error}} \qquad \textsc{D-seq}\ \frac{\Omega_{c,0} \vdash c \Rrightarrow \Omega_{c,1} \qquad \Omega_{c,1} \vdash P \Rrightarrow \Omega_{c,2}}{\Omega_{c,0} \vdash c\ ;\ P \Rrightarrow \Omega_{c,2}}$$

$$\textsc{D-assign-table}\ \frac{\Omega_c \neq \texttt{error} \qquad \Omega_c[\texttt{X} \mapsto 0] \vdash e \Downarrow v_0 \qquad \cdots \qquad \Omega_c[\texttt{X} \mapsto n-1] \vdash e \Downarrow v_{n-1}}{\Omega_c \vdash x[\texttt{X}, n] := e \Rrightarrow \Omega_c[x \mapsto (v_0, \ldots, v_{n-1})]}$$

Figure 5. Operational semantics: statements

$e_1 \odot e_2, \odot \in \{+, -\}$	undef	$f_2 \in \mathbb{F}$
undef	undef	$0 \odot f_2$
$f_1 \in \mathbb{F}$	$f_1 \odot 0$	$f_1 \odot_{\mathbb{F}} f_2$

$e_1 \odot e_2, \odot \in \{\times, \div\}$	undef	$f_2 \in \mathbb{F}, f_2 \neq 0$	0
undef	undef	undef	undef
f_1	undef	$f_1 \odot_{\mathbb{F}} f_2$	0

b_1 ⟨*boolop*⟩ b_2	undef	$f_2 \in \mathbb{F}$
undef	undef	undef
$f_1 \in \mathbb{F}$	undef	f_1 ⟨*boolop*⟩$_{\mathbb{F}}$ f_2

$m(e_1, e_2), m \in \{\texttt{min}, \texttt{max}\}$	undef	$f_2 \in \mathbb{F}$
undef	0	$m_{\mathbb{F}}(0, f_2)$
$f_1 \in \mathbb{F}$	$m_{\mathbb{F}}(f_1, 0)$	$m_{\mathbb{F}}(f_1, f_2)$

round(undef) = undef
round($f \in \mathbb{F}$) = $\text{floor}_{\mathbb{F}}(f + \texttt{sign}(f) * 0.50005)$
truncate(undef) = undef
truncate($f \in \mathbb{F}$) = $\text{floor}_{\mathbb{F}}(f + 10^{-6})$
abs(x) ≡ if x >= 0 then x else -x
pos_or_null (x) ≡ x >= 0
pos(x) ≡ x > 0
null(x) ≡ x = 0
present(undef) = 0
present($f \in \mathbb{F}$) = 1

Figure 6. Function semantics. For context on `round` and `truncate` definitions, see Section 4.3

⟨*program*⟩ ::= ⟨*fundecl*⟩*

⟨*fundecl*⟩ ::= ⟨*funname*⟩ (⟨*var*⟩*) : ⟨*command*⟩*

⟨*command*⟩ ::= if ⟨*expr*⟩ then ⟨*command*⟩* else ⟨*command*⟩*
| partition with ⟨*var_kind*⟩ : ⟨*command*⟩*
| ⟨*var*⟩ = ⟨*expr*⟩ | ⟨*var*⟩* <- ⟨*fun*⟩() | del ⟨*var*⟩

⟨*expr*⟩ ::= ⟨*var*⟩ | ⟨*float*⟩ | undef | ⟨*expr*⟩ ⟨*binop*⟩ ⟨*expr*⟩ | ⟨*unop*⟩ ⟨*expr*⟩
| ⟨*builtin*⟩ (⟨*expr*⟩, ..., ⟨*expr*⟩) | exists(⟨*var_kind*⟩)

⟨*binop*⟩ ::= ⟨*arithop*⟩ | ⟨*boolop*⟩

⟨*arithop*⟩ ::= + | - | * | /

⟨*boolop*⟩ ::= <= | < | > | >= | == | != | && | ||

⟨*unop*⟩ ::= - | ~

⟨*var_kind*⟩ ::= taxbenefit | deposit | ...

⟨*fun*⟩ ::= ⟨*funname*⟩ | call_m

⟨*builtin*⟩ ::= present | cast

Figure 7. Syntax of the M++ language

undefined variables, this provides an additional level of guarantee, by ensuring that reduction always produces a value or an error (i.e. we haven't forgotten any corner cases in our semantics). Furthermore, we show in the process that the store is consistent with the typing environment, written $\Gamma \triangleright \Omega$. This entails store typing (i.e. values of the right type are to be found in the store) and proper handling of undefined variables (i.e. dom $\Omega \subseteq$ dom Γ).

Theorem (Expressions)**.** If $\Gamma \triangleright \Omega$ and $\Gamma \vdash e$, then there exists v such that $\Gamma \vdash e \Downarrow v$.

We extend $\triangleright$ to statements, so as to account for exceptions:

$$\Gamma \triangleright_c \Omega_c \Longleftrightarrow \Omega_c = \texttt{error} \vee \Gamma \triangleright \Omega_c$$

Theorem (Statements)**.** If $\Gamma \vdash c \Rrightarrow \Gamma'$ et $\Gamma \triangleright_c \Omega_c$, then there exists Ω'_c such that $\Omega_c \vdash c \Rrightarrow \Omega'_c$ and $\Gamma' \triangleright_c \Omega'_c$.

We provide full proofs and definitions in Coq, along with a guided tour of our development, in the supplement [21].

3 The Design of a New DSL: M++

As described in Fig. 1, the internal compiler of the DGFiP compiles M files (Section 2) to C code. Insofar as we understand, the M codebase originally expressed the whole income tax computation. However, in the 1990s (Section 1), the DGFiP started executing the M code twice, with slightly different parameters, in order for the taxpayer to witness the impact of a tax reform. Rather than extending M with support for user-defined functions, the DGFiP wrote the new logic in C, in a folder called "inter", for multi-year computations. This piece of code can read and write variables used in the M codebase using shared global state. To assemble the final executable, M-produced C files and hand-written "inter" C files are compiled by GCC and distributed as a shared library. Over time, the "inter" folder grew to handle a variety of special cases, multiplying calls into the M codebase. At the time of writing, the "inter" folder amounts to 35,000 lines of C code.

This poses numerous problems. First, the mere fact that "inter" is written in C prevents it from being released to the public, the DGFiP fearing security issues that might somehow be triggered by malicious inputs provided by the taxpayer. Therefore, the taxpayer cannot reproduce the tax computation since key parts of the logic are missing. Second, by virtue of being written in C, "inter" does not compose with M, hindering maintainability, readability and auditability. Third, C limits the ability to modernize the codebase; right now, the online tax simulator is entirely written in C using Apache's CGI feature (including HTML code generation), a very legacy infrastructure for Web-based development. Fourth, C is notoriously hard to analyze, preventing both the DGFiP and the taxpayer from doing fine-grained analyses.

To address all of these limitations, we design M++, a companion domain-specific language (DSL) that is powerful enough to completely eliminate the hand-written C code.

3.1 Concrete syntax and new constructions

The chief purpose of the M++ DSL is to repeatedly call the M rules, with different M variable assignments for each call. To assist with this task, M++ provides basic computational

facilities, such as functions and local variables. In essence, M++ allows implementing a "driver" for the M code.

Fig. 8 shows concrete syntax for M++. We chose syntax resembling Python, where block scope is defined by indentation. As the French administration moves towards a modern digital infrastructure, Python seems to be reasonably understood across various administrative services.

Fig. 7 formally lists all of the language constructs that M++ provides. A program is a sequence of function declarations. M++ features two flavors of variables. Local variables follow scoping rules similar to Python: there is one local variable scope per function body; however, unlike Python, we disallow shadowing and have no block scope or `nonlocal` keyword. Local variables exist only in M++. Variables in all-caps live in the M variable scope, which is shared between M and M++, and obey particular semantics.

3.2 Semantics of M++

Two constructs support the interaction between M and M++: the `<-` and `partition` operators. They have slightly unusual semantics, in the way that they deal with the M variable scope. These semantics are heavily influenced by the needs of the DGFiP, as we strived to provide something that would feel intuitive to technicians in the French administration.

To precisely define the expected behavior, Fig. 9 presents reduction semantics of the form $\Delta, \Omega_1 \vdash c \rightsquigarrow \Omega_2$, meaning command c updates the store from Ω_1 to Ω_2, given the functions declared in Δ.

We distinguish built-ins, which may only appear in expressions and do not modify the global store, from functions, which are declared at the top-level and may modify the store. The `call_m` operation is a special function. The `<-` operator takes a *function* call, and executes it in a copy of the memory. Then, only those variables that appear on the left-hand side see their value propagated to the parent execution environment. Thus, `call_m` only affects variables $\vec{X}$.

To execute the function call, the `<-` operator either looks up definitions in Δ, the environment of user-defined functions, or executes the M rules in the `call_m` case, relying on the earlier definition of $\Rrightarrow$ (Fig. 5).

Worded differently, our semantics introduce a notion of call stack and treat the M computation as a function call returning multiple values. It is to be noted that the original C code had no such notion, and that the $\vec{X}$ were nothing more than mere comments. As such, there was no way to statically rule out potential hidden state persisting from one `call_m` to another since the global scope was modified in place. With this formalization and its companion implementation (Section 4), we were able to confirm that there is currently no reliance on hidden state (something which we suspect took considerable effort to enforce in the hand-written C code), and were able to design a much more principled semantics that we believe will lower the risk of future errors.

```
1  compute_benefits():
2   if exists(taxbenefit) or exists(deposit):
3      V_INDTEO = 1
4      V_CALCUL_NAPS = 1
5      partition with taxbenefit:
6        NAPSANSPENA, IAD11, INE, IRE, PREM8_11 <- call_m()
7      iad11 = cast(IAD11)
8      ire = cast(IRE)
9      ine = cast(INE)
10     prem = cast(PREM8_11)
11     V_CALCUL_NAPS = 0
12     V_IAD11TEO = iad11
13     V_IRETEO = ire
14     V_INETEO = ine
15     PREM8_11 = prem
```

Figure 8. Example function defined in M++

The `partition` operation operates over a variable kind k (Section 2.1). The sub-block c of `partition` executes in a restricted scope, where variables having kind k are temporarily set to `undef`. Upon completion of c, the variables at kind k are restored to their original value, while other variables are propagated from the sub-computation into the parent scope. This allows running computations while "disabling" groups of variables, e.g. ignoring an entire category of tax credits.

3.3 Example

Fig. 8 provides a complete M++ example, namely the function `compute_benefits`.

The conditional at line 2 uses a variable kind-check (Section 2.1) to see if any variables of kind "tax benefit" have a non-`undef` value. Then, lines 3-4 set some flags before calling M. Line 5 tells us that the call to M at line 6 is to be executed in a restricted context where variables of kind "tax benefit" are set to `undef`. Line 6 runs the M computation, over the current state of the M variables; five M output variables are retained from this M execution, while the rest are discarded. Lines 7-11 represent local variable assignment, where `cast` has the same effect as `+ 0` in M, namely, forcing the conversion of `undef` to 0. Then, lines 11-15 set M some variables as input for later function calls.

4 Mlang: an M/M++ Compiler

After clarifying the semantics of M (Section 2), and designing a new DSL to address its shortcomings (M++, Section 3), we now present Mlang, a modern compiler for both M and M++.

4.1 Architecture of Mlang

Mlang takes as input an M codebase, an M++ file, and a file specifying assumptions (described in the next paragraph). Mlang currently generates Python or C; it also offers a built-in interpreter for computations. Mlang is implemented in OCaml, with around 9,000 lines of code. The general architecture is shown in Fig. 10. The M files and the M++ program

Judgments: $\boxed{\Delta, \Omega \vdash e \wr v}$ ("Under Δ, Ω, e evaluates into v") $\boxed{\Delta, \Omega_1 \vdash c \rightsquigarrow \Omega_2}$ ("Under Δ, c transforms Ω_1 into Ω_2")

$$\textsc{Cast-float}\quad \frac{\Delta, \Omega \vdash \mathsf{e} \wr f \qquad f \neq \mathsf{undef}}{\Delta, \Omega \vdash \mathsf{cast(e)} \wr f}$$

$$\textsc{Cast-undef}\quad \frac{\Delta, \Omega \vdash \mathsf{e} \wr \mathsf{undef}}{\Delta, \Omega \vdash \mathsf{cast(e)} \wr 0}$$

$$\textsc{Exists-true}\quad \frac{\exists X \in \Omega, kind(X) = k \wedge \Omega(X) \neq \mathsf{undef}}{\Delta, \Omega \vdash \mathsf{exists(k)} \wr 1}$$

$$\textsc{Exists-false}\quad \frac{\forall X \in \Omega, kind(X) \neq k \vee \Omega(X) = \mathsf{undef}}{\Delta, \Omega \vdash \mathsf{exists(k)} \wr 0}$$

$$\textsc{Call}\quad \frac{\begin{array}{lll} \Omega_1 \vdash \mathsf{M\ rules} \Rrightarrow \Omega_2 & \text{if f = call_m} & \Omega_3(Y) = \Omega_1(Y) \text{ if } Y \notin \vec{X} \\ \Delta, \Omega_1 \vdash \Delta(f) \rightsquigarrow \Omega_2 & \text{otherwise} & \Omega_3(Y) = \Omega_2(Y) \text{ if } Y \in \vec{X} \end{array}}{\Delta, \Omega_1 \vdash \vec{X} \leftarrow \mathsf{f}() \rightsquigarrow \Omega_3}$$

$$\textsc{Partition}\quad \frac{\begin{array}{l} \Omega_2(Y) = \mathsf{undef} \text{ if } \mathsf{kind}(Y) = k \\ \Omega_2(Y) = \Omega_1(Y) \text{ otherwise} \end{array} \quad \Delta, \Omega_2 \vdash c \rightsquigarrow \Omega_3 \quad \begin{array}{l} \Omega_4(Y) = \Omega_1(Y) \text{ if } \mathsf{kind}(Y) = k \\ \Omega_4(Y) = \Omega_3(Y) \text{ otherwise} \end{array}}{\Delta, \Omega_1 \vdash \mathsf{partition\ with\ k} : c \rightsquigarrow \Omega_4}$$

$$\textsc{Delete}\quad \frac{\Delta, \Omega_1 \vdash \mathsf{v = undef} \rightsquigarrow \Omega_2}{\Delta, \Omega_1 \vdash \mathsf{del\ v} \rightsquigarrow \Omega_2}$$

Figure 9. Reduction rules of M++

are first parsed and transformed into intermediate representations. These intermediate representations are inlined into a single backend intermediate representation (BIR), consisting of assignments and conditionals. Inlining is aware of the semantic subtleties described in Fig. 9 and uses temporary variable assignments to save/restore the shared M/M++ scope. BIR code is then translated to the optimization intermediate representation (OIR) in order to perform optimizations. OIR is the control-flow-graph (CFG) equivalent of BIR.

OIR is the representation on which we perform our optimizations (Section 4.2). For instance, in order to perform constant propagation, we must check that a given assignment to a variable dominates all its subsequent uses. A CFG is the best data structure for this kind of analysis. We later on switch back to the AST-based BIR in order to generate textual C output.

Additional assumptions. In M, a variable not defined in the current memory environment evaluates to `undef` (rule D-Var-Undef, Fig. 4). This permissive behavior is fine for an interpreter which has a dynamic execution environment; however, our goal is to generate efficient C and Python code that can be integrated into existing software. As such, declaring every single one of the 27,113 possible variables (as found in the original M rules) in C would be quite unsavory.

We therefore devise a mechanism that allows stating ahead of time which variables can be truly treated as inputs, and which are the outputs that we are interested in. Since these vary depending on the use-case, we choose to list these assumptions in a separate file that can be provided alongside with the M/M++ source code, rather than making this an intrinsic, immutable property set at variable-declaration time. Doing so increases the quality of the generated C or Python.

We call these *assumption files*; we have hand-written 5 of those. **All** is the empty file, i.e. no additional assumptions. This leaves 2459 input variables, and 10,411 output variables for the 2018 codebase. **Selected outs** enables all input variables, but retains only 11 output variables. **Tests** corresponds to the inputs and outputs used in the test files used by the DGFiP. **Simplified** corresponds to the simplified simulator released each year by the DGFiP a few months before the full income tax computation is released. There are 214 inputs, and we chose 11 output variables. **Basic** accepts as inputs only the marital status and the salaries of each individual of the couple. The output is the income tax.

4.2 Optimizations

In the 2018 tax code, the initial number of BIR instructions after inlining M and M++ files together is 656,020. This essentially corresponds to what the legacy compiler normally generates, since it performs no optimizations.

Thanks to its modern compiler architecture, Mlang can easily perform numerous textbook optimizations, namely dead code elimination, inlining and partial evaluation. This allows greatly improving the quality of the generated code.

We now present a series of optimizations, performed on the OIR intermediate representation. The number of instructions after these optimizations is shown in Fig. 11. Without any assumption (**All**), the optimizations shrink the generated C code to 15% size (a factor of 6.5). With the most restrictive assumption file (**Simplified**), only 0.47% optimization.

Definedness analysis. Due to the presence of `undef`, some usual optimizations are not available. For example, optimizing `e * 0` into `0` is incorrect when `e` is `undef`, as `undef * 0 = undef`. Similarly, `e + 0` cannot be rewritten as `e`. Our partial evaluation is thus combined with a simple definedness analysis. The lattice of the analysis is shown in Fig. 12; we use the standard sharp symbol of abstract interpretation [7] to denote abstract elements. The transfer function $\mathtt{absorb}^{\#}$ defined in Fig. 13 is used to compute the definedness in the case of the multiplication, the division and all operators in $\langle boolop \rangle$. The $\mathtt{cast}^{\#}$ transfer function is used for the addition and the subtraction.

This definedness analysis enables finer-grained partial evaluation rules, such as those presented in Fig. 14.

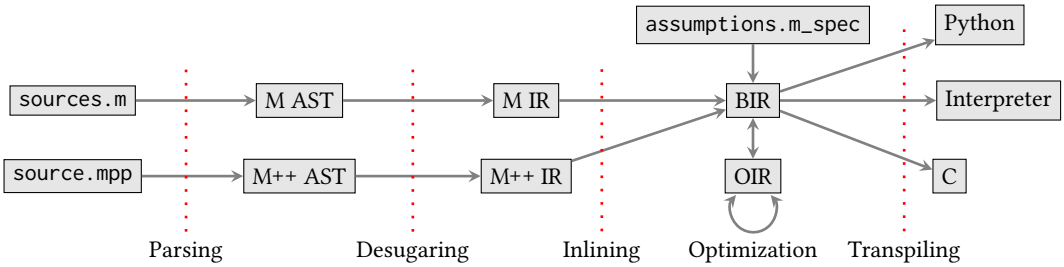

Figure 10. MLANG compilation passes

Spec. name	# inputs	# outputs	# instructions
All	2,459	10,411	129,683
Selected outs	2,459	11	99,922
Tests	1,635	646	111,839
Simplified	228	11	4,172
Basic	3	1	553

Figure 11. Number of instructions generated after optimization. Instructions with optimizations disabled: 656,020.

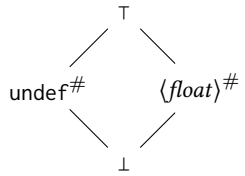

Figure 12. Definedness lattice

d_1	d_2	$\texttt{absorb}^{\#}(d_1, d_2)$	$\texttt{cast}^{\#}(d_1, d_2)$
$\texttt{undef}^{\#}$	$\texttt{undef}^{\#}$	$\texttt{undef}^{\#}$	$\texttt{undef}^{\#}$
$\texttt{undef}^{\#}$	$\langle float \rangle^{\#}$	$\texttt{undef}^{\#}$	$\langle float \rangle^{\#}$
$\langle float \rangle^{\#}$	$\texttt{undef}^{\#}$	$\texttt{undef}^{\#}$	$\langle float \rangle^{\#}$
$\langle float \rangle^{\#}$	$\langle float \rangle^{\#}$	$\langle float \rangle^{\#}$	$\langle float \rangle^{\#}$

Figure 13. Transfer functions over the definedness lattice, implicitly lifted to the full lattice.

$$
\begin{array}{ll}
e + \texttt{undef} \rightsquigarrow e & e : \langle float \rangle^{\#} + 0 \rightsquigarrow e \\
e * 1 \rightsquigarrow e & e : \langle float \rangle^{\#} * 0 \rightsquigarrow 0 \\
\texttt{max}(0, \texttt{min}(0, x)) \rightsquigarrow 0 & \texttt{present}(\texttt{undef}) \rightsquigarrow 0 \\
\texttt{max}(0, -\texttt{max}(0, x)) \rightsquigarrow 0 & \texttt{present}(e : \langle float \rangle^{\#}) \rightsquigarrow 1
\end{array}
$$

Figure 14. Examples of optimizations

The optimizations for `+ 0` and `* 0` are invalid in the presence of IEEE-754 special values (NaN, minus zero, infinities) [3, 23]. We have instrumented the M code to confirm that these are valid on the values used. But for safety, these unsafe optimizations are only enabled if the `--fast_math` flag is set.

```
// my_var1 is a local variable always defined
#define my_truncate(a) ( my_var1=(a)+0.000001,floor(my_var1) )
#define my_round(a) (floor(
  (a<0) ? (double)(long long)(a-.50005)
        : (double)(long long)(a+.50005)))
```

Figure 15. Custom rounding and truncation rules

4.3 Backends

DGFiP (legacy). The DGFiP's legacy system has a single backend that produces pre-ANSI (K&R) C. For each M rule, two C computations are emitted. The first one aims to determine whether the resulting value is defined. It operates on C's `char` type, where `0` is undefined or `1` is defined. The second computation is syntactically identical, except it operates on `double` and thus computes the actual arithmetic expression. This two-step process explains some of the operational semantics: with `0` being undefined, the special value `undef` is absorbing for e.g. the multiplication.

Careful study of the generated code also allowed us to nail down some non-standard rounding and truncation rules which had until then eluded us. We list them in Fig. 15; these are used to implement the built-in operators from Fig. 2 in both our interpreter and backends.

MLANG. Our backend generates C and Python from BIR. Since BIR only features assignments, arithmetic and conditionals, we plan to extend it with backends for JavaScript, R/MatLab and even SQL for in-database native tax computation. Depending on the DGFiP's appetite for formal verification, we may verify the whole compiler since the semantics are relatively small.

Implementing a new backend is not very onerous: it took us 500 lines for the C backend and 375 lines for the Python backend. Both backends are validated by running them over the entire test suite and comparing the result with our reference interpreter.

Our generated code only relies on a small library of helpers which implement operations over M values. These helpers

Scheme	M compiler	C compiler	Bin. size	Time
Original	DGFiP	GCC `-O0`	7 Mo	~ 1.5 ms
Original	DGFiP	GCC `-O1`	7 Mo	~ 1.5 ms
Array	Mlang	Clang `-O0`	19 Mo	~ 4 ms
Array	Mlang	Clang `-O1`	10 Mo	~ 2 ms

Figure 16. Performance of the C code generated by various compilation schemes for the M code. The time measured is the time spent inside the main tax computation function for one fiscal household picked in the set of test cases. Size of the compiled binary is indicated. "Original" corresponds to the DGFiP's legacy system. "Local vars" corresponds to Mlang's C backend mapping each M variable to a C local variable.

are aware of all the semantic subtleties of M and are manually audited against the paper semantics.

5 Analyzing and Evaluating the Tax Code

Due to the sheer size of the code and number of variables, generating efficient code is somewhat delicate – we had the pleasure of breaking both the Clang and Python parsers because of an exceedingly naïve translation. Thankfully, owing to our flexible architecture for Mlang, we were able to quickly iterate and evaluate several design choices.

We now show the benefits of a modern compiler infrastructure, and proceed to describe a variety of instrumentations, techniques and tweaking knobs that allowed us to gain insights on the the tax computation. By bringing the M language into the 21st century, we not only greatly enhance the quality of the generated code, but also unlock a host of techniques that significantly increase our confidence in the French tax computation.

5.1 Performance of the generated code

We initially generated C code that would emit one local variable per M variable. But with tens of thousands of local variables, running the code required `ulimit -s`.

We analyzed the legacy code and found out that the DGFiP stored all of the M variables in a global array. We implemented the same technique and found out that with `-O1`, we were almost as fast as the legacy code. We attribute this improvement to the fact that the array, which is a few dozen kB, which fits in the L2 cache of most modern processors. This is a surprisingly fortuitous choice by the DGFiP. See Fig. 16 for full results. In the grand scheme of things, the cost of computing the final tax is dwarfed by the time spent generating a PDF summary for the taxpayer (~200ms). The 500µs difference between the DGFiP's system and ours is thus insignificant.

5.2 The cost of IEEE-754

Relying on IEEE-754 and its limited precision for something as crucial as the income tax of an entire nation naturally raises questions. Thanks to our new infrastructure, we were able to instrument the generated code and gain numerous insights.

Does precision matter? We tweaked our backend to use the MPFR multiprecision library [13]. With 1024-bit floats, all tests still pass, meaning that there is no loss of precision with the double-precision 64-bit format.

Does rounding matter? We then instrumented the code to measure the effect of the IEEE-754 rounding mode on the final result. Anything other than the default (rounding to nearest, ties to even) generates incorrect results. The control-flow remains roughly the same, but some comparisons against 0 do give out different results as the computation skews negatively or positively. We plan in the future to devise a static analysis that could formally detect errors, such as comparisons that are always false, or numbers that may be suspiciously close to zero (denormals).

Fixed precision. Nevertheless, floating-point computations are notoriously hard to analyze and reason about, so we set out to investigate replacing floats with integer values. In our first experiment, we adopted big decimals, i.e. a bignum for the integer part and a fixed amount of digits for the fractional part. Our test suite indicates that the integer part never exceeds 9999999999 (encodable in 37 bits); it also indicates that with 40 bits of precision for the fractional part, we get correct results. This means that a 128-bit integer would be a viable alternative to a `double`, with the added advantage that formal analysis tools would be able to deal with it much better.

Using rationals. Finally, we wondered if it was possible to completely work without floating-point and eliminate imprecision altogether, taking low-level details such as rounding mode and signed zeroes completely out of the picture.

To that end, we encoded values as fractions where both numerator and denominator are big integers. We observed that both never exceed 2^{128}, meaning we could conceivably implement values as a struct with two 128-bit integers and a sign bit. We have yet to investigate the performance impact of this change.

5.3 Test-case generation

The DGFiP test suite is painstakingly constructed by hand by lawyers year after year. From this test suite, we extracted 476 usable test cases that don't raise any exceptions (see Section 2.1). The DGFiP has no infrastructure to automatically generate cases that would exercise new situations. As such, the test suite remains relatively limited in the variety of households it covers. Furthermore, many of the hand-written tests are for previous editions of the tax code, and describe situations that would be rejected by the current tax code.

Generating test cases is actually non-trivial: the search space is incredibly large, owing to the amount of variables, but also deeply constrained, owing to the fact that most

variables only admit a few possible values (Section 1), and are further constrained in relationship to other variables.

We now set out to automatically generate fresh (valid) test cases for the tax computation, with two objectives: assert on a very large number of test cases that our code and the legacy implementation compute the same result; and exhibit corner cases that were previously not exercised, so as to generate fresh novel tax situations for lawmakers to consider.

Randomized testing. We start by randomly mutating the legacy test suite, in order to generate new distinct, valid test cases. If a test case raises an exception, we discard it. We obtain 1267 tests, but these are, unsurprisingly, very close to the legacy test suite and do not exercise very many new situations. They did, however, help us when reverse-engineering the semantics of M. We now have 100% conformance on those tests.

Coverage-guided fuzzing. In order to better explore the search space, we turn to AFL [33]. The tool admits several usage modes – finding genuine crashes (e.g. segfaults), or generating test cases for further seeding into the rest of the testing pipeline. We focus on the latter mode, meaning that we generate an artificial "crash" when a synthesized testcase raises no M errors, that is, when we have found a valid testcase. We first devise an injection from opaque binary inputs, which AFL controls, to the DGFiP input variables. Once "crashes" have been collected, we simply emit a set of test inputs that has the same format as the DGFiP.

Thanks to this very flexible architecture, we were able to perform fully general fuzzing exercising all input variables, as well as targeted fuzzing that focuses on a subset of the variables. The former takes a few hours on a high-end machine; the latter mere minutes. We synthesized around 30,000 tests cases, which we reduced down to 275 using afl-cmin.

So far, the fuzzer-generated test case have pointed out of a few bugs in Mlang's optimizations and backends. We plan to further use AFL to find find test cases that satisfy extra properties not originally present in the tax code, e.g. an excessively high marginal tax rate that might raise some legality questions.

Symbolic execution fuzzing. We attempted to use dynamic symbolic execution tool KLEE [5], but found out that it only had extremely limited support for floating-point computations. As detailed earlier (Section 5.2), we have found that integer based computations are a valid replacement for floats, and plan to use this alternate compilation scheme to investigate whether KLEE would provide interesting test cases.

5.4 Coverage measurements

Finally, we wish to evaluate how "good" our new test cases are. Code coverage seems like a natural notion, especially seeing that there is currently none in the DGFiP infrastructure. However, traditional code coverage makes little sense: conditionals are very rare in the generated code.

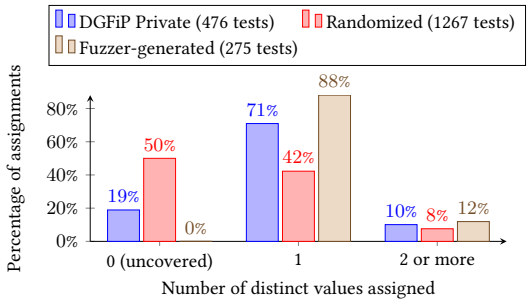

Figure 17. Value coverage of assignments for each test suite

Rather, we focus on value coverage: for each assignment in the code, we count the number of distinct values assigned during the execution of an entire test case. This is a good proxy for test quality: the more different values flow through an assignment, the more interesting the tax situation is.

Fig. 17 shows our measurements. The first take-away is that our randomized tests did not result in meaningful tests: the number of assignments that are uncovered actually increased. The tests we obtained with AFL, however, significantly increase the quality of test coverage. We managed to synthesize many tests that exercise statements previously unvisited by the DGFiP's test suite, and exhibit much more complex assignments (2 or more different values assigned).

Our knowledge of the existing DGFiP test suite is incomplete, as we only have access to a partial set of tests. In particular, a special set of rules apply when the tax needs to be adjusted later on following an audit, and the tests for these have not been communicated to us. We hope to obtain visibility onto those in the future.

6 Related Work and Conclusion

6.1 Implementing the law

Formalizing part of the law using logic programming or a custom domain specific language has been extensively tried in the past, as early as 1914 [1, 2, 9, 12, 15, 25, 28]. Most of these works follow the same structure: they take a subset of the law, analyze its logical structure, and encode it using a novel or existing formalism. All of them stress the complexity of this formal endeavor, coming from *i)* the underlying reality that the law models and *ii)* the logical structure of the legislative text itself. After more a century of research, no silver bullet has emerged that would allow to systematically translate the text of a law into a formal model.

However, domain-specific attempts have been more successful. Recently, blockchain has demonstrated increased interest for domain-specific languages encoding smart contracts [14, 16, 27, 32]. Regular private commercial contracts have also been targeted for formalization [6, 30], as well

as financial contracts [11, 24]. Concerning the public sector, the "rules as code" movement has been the object of an exhaustive OECD report [22].

Closer to the topic of this paper, the logical structure of the US tax law has been extensively studied by Lawsky [18, 19], pointing out the legal ambiguities in the text of the law that need to be resolved using legal reasoning. She also claims that the tax law drafting style follows default logic [26], a non-monotonic logic that is hard to encode in languages with first-order logic (FOL). This could explain, as M is also based on FOL, the complexity of the DGFiP codebase.

As this complexity generates opacity around the way taxes are computed, another government agency set out to re-implement the entire French socio-fiscal system in Python [29]. Even if this initiative was helpful and used as a computation backend for various online simulators, the results it returns are not legally binding, unlike the results returned by the DGFiP. Furthermore, this Python implementation does not deal with all the corner cases of the law. To the extent of our knowledge, our work is unprecedented in terms of size and exhaustiveness of the portion of the law turned into a reusable and formalized software artifact.

6.2 Conclusion

Thanks to modern compiler construction techniques, we have been able to lift up a legacy, secret codebase into a reusable, public artifact that can be distributed into virtually any programming environment. The natural next step for the DGFiP is to consider taking more insight from programming languages research, and design a successor to M/M++ that provides good tooling for translating the tax law into a correct and distributable implementation.

Acknowledgments

This work is partially supported by the European Research Council under Consolidator Grant Agreement 681393 — MOPSA and 683032 — CIRCUS.

References

[1] Layman E Allen. 1956. Symbolic logic: A razor-edged tool for drafting and interpreting legal documents. *Yale LJ* 66 (1956), 833.

[2] Layman E. Allen and C. Rudy Engholm. 1978. Normalized legal drafting and the query method. *Journal of Legal Education* 29, 4 (1978), 380–412.

[3] Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. 2015. Verified Compilation of Floating-Point Computations. *J. Autom. Reason.* 54, 2 (2015), 135–163. https://doi.org/10.1007/s10817-014-9317-x

[4] Sylvie Boldo and Guillaume Melquiond. 2011. Flocq: A Unified Library for Proving Floating-Point Algorithms in Coq. In *20th IEEE Symposium on Computer Arithmetic, ARITH 2011, Tübingen, Germany, 25-27 July 2011*, Elisardo Antelo, David Hough, and Paolo Ienne (Eds.). IEEE Computer Society, 243–252. https://doi.org/10.1109/ARITH.2011.40

[5] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.. In *OSDI*, Vol. 8. 209–224.

[6] John J Camilleri. 2017. *Contracts and Computation—Formal modelling and analysis for normative natural language.*

[7] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*. ACM, 238–252.

[8] Equipe Leximpact de L'Assemblée nationale. 2019. *LexImpact.* https://leximpact.an.fr/

[9] John Dewey. 1914. Logical method and law. *Cornell LQ* 10 (1914), 17.

[10] Direction Générale des Finances Publiques (DGFiP). 2019. *Les règles du moteur de calcul de l'impôt sur le revenu et de l'impôt sur la fortune immobilière.* https://gitlab.adullact.net/dgfip/ir-calcul

[11] Jean-Marc Eber. 2009. The Financial Crisis, a Lack of Contract Specification Tools: What Can Finance Learn from Programming Language Design?.. In *ESOP*. 205–206.

[12] D Fernández Duque, M González Bedmar, D Sousa, Joosten, J.J, and G. Errezil Alberdi. 2019. 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.* TransJus Working Papers Publication - Edición Especial, 159–171. http://diposit.ub.edu/dspace/handle/2445/137759

[13] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. 2007. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software (TOMS)* 33, 2 (2007), 13–es.

[14] Xiao He, Bohan Qin, Yan Zhu, Xing Chen, and Yi Liu. 2018. Spesc: A specification language for smart contracts. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. IEEE, 132–137.

[15] Nils Holzenberger, Andrew Blair-Stanek, and Benjamin Van Durme. 2020. A Dataset for Statutory Reasoning in Tax Law Entailment and Question Answering. *arXiv preprint arXiv:2005.05257* (2020).

[16] Tom Hvitved. 2011. *Contract formalisation and modular implementation of domain-specific languages.* Ph.D. Dissertation. Citeseer.

[17] Camille Landais, Thomas Piketty, and Emmanuel Saez. 2011. *Pour une révolution fiscale: Un impôt sur le revenu pour le 21 ème siècle.* https://www.revolution-fiscale.fr/simuler/irpp/

[18] Sarah B. Lawsky. 2017. Formalizing the Code. *Tax Law Review* 70, 377 (2017).

[19] Sarah B. Lawsky. 2018. A Logic for Statutes. *Florida Tax Review* (2018).

[20] Denis Merigoux, Raphaël Monat, and Jonathan Protzenko. 2020. *Mlang, A Modern Compiler for the French Tax Code.* https://github.com/MLanguage/mlang

[21] Denis Merigoux, Raphaël Monat, and Jonathan Protzenko. 2021. *A Modern Compiler for the French Tax Code - Artifact.* https://doi.org/10.5281/zenodo.4456774

[22] James Mohun and Alex Roberts. 2020. Cracking the code: Rulemaking for humans and machines. (2020).

[23] Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. 2018. *Handbook of Floating-Point Arithmetic (2nd Ed.).* Springer. https://doi.org/10.1007/978-3-319-76526-6

[24] Grant Olney Passmore and Denis Ignatovich. 2017. Formal Verification of Financial Algorithms. In *CADE.* https://doi.org/10.1007/978-3-319-63046-5_3

[25] Marcos A Pertierra, Sarah Lawsky, Erik Hemberg, and Una-May O'Reilly. 2017. Towards Formalizing Statute Law as Default Logic through Automatic Semantic Parsing.. In *ASAIL@ ICAIL.*

[26] R. Reiter. 1987. Readings in Nonmonotonic Reasoning. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, Chapter A Logic for Default Reasoning, 68–93. http://dl.acm.org/citation.cfm?id=42641.42646

[27] Vincenzo Scoca, Rafael Brundo Uriarte, and Rocco De Nicola. 2017. Smart contract negotiation in cloud computing. In *2017 IEEE 10th*

International Conference on Cloud Computing (CLOUD). IEEE, 592–599.

[28] M. J. Sergot, F. Sadri, R. A. Kowalski, F. Kriwaczek, P. Hammond, and H. T. Cory. 1986. The British Nationality Act As a Logic Program. *Commun. ACM* 29, 5 (May 1986), 370–386.

[29] Sébastien Shulz. 2019. Free software to tackle the lack of transparency in the social tax system. Sociology of a heterogeneous movement at the margins of the state. *Revue francaise de science politique* 69, 5 (2019), 845–868.

[30] SMU Centre for Computational Law. 2020. *The L4 domain-specific language*. https://github.com/smucclaw/dsl

[31] The Coq Development Team. 2017. The Coq Proof Assistant Reference Manual, version 8.7. http://coq.inria.fr

[32] Jakub Zakrzewski. 2018. Towards verification of Ethereum smart contracts: a formalization of core of Solidity. In *Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer, 229–247.

[33] Michal Zalewski. 2014. American fuzzy lop.