Catala: Moving Towards the Future of Legal Expert Systems

Liane Huttner, Denis Merigoux

Noname manuscript No.
(will be inserted by the editor)

CATALA: Moving Towards the Future of Legal Expert Systems

Liane Huttner · **Denis Merigoux**

Received: date / Accepted: date

Abstract Around the world, private and public organizations use software called legal expert systems to compute taxes. This software must comply with the laws they are designed to implement. As such, a bug or an error in a program that leads to tax miscalculations can have heavy legal and democratic consequences. However, increasing evidence suggests that some legal expert systems may not comply with the law. Moreover, traditional software development processes mean that legal expert systems are difficult to adapt to the continuous flow of new legislation. To prevent further software decay and to reconcile these systems with the growing demand for algorithmic transparency, we argue that there is a need for a new development process for legal expert systems. This new system must be built to comply with the law, in particular the GDPR. It must also respect democratic transparency. For these reasons, we present a solution built by lawyers and computer scientists: CATALA, a new programming language coupled with a pair programming development process.

Keywords legal expert systems, formal methods, literate programming, algorithmic transparency, tax law, social benefits

Liane Huttner
Université Panthéon-Sorbonne, Paris
E-mail: huttner.liane@gmail.com

Denis Merigoux
Inria, Paris
E-mail: denis.merigoux@inria.fr

1 Introduction

How do governments compute taxes and social benefits for their citizens? Many have chosen to solve this complicated task by using computer programs. They have created software, modeled on the law, which computes the amount of tax due and the social benefits to be paid to each citizen.

These programs, called "legal expert systems", are based on the law. In other words, the program and its reference statute ought to be functionally equivalent. This means that any result output by the computer program should match a correct legal reasoning based on the statute. A breach of functional equivalence in such programs, which we call a bug, places the entity running the program in legal jeopardy.

Two questions arise from this situation. The first one is legal and social: what are the consequences of bugs in programs computing taxes? The second is technical: how to decrease the number of bugs in those programs, and increase the confidence that they faithfully implement the statutes they use as a reference?

We will answer those questions with a survey of the state of the art of the computerized implementation of tax-related statutes, with a focus on French examples (§2). Then, we will present the legal and democratic implications of bugs in legal expert systems (§3). Finally, we will introduce CATALA, a new programming language created by lawyers and computer scientists for quantitative statute formalization. The goal of CATALA is to provide a systematic way to produce bug-free programs from tax-related statutes, that can be deployed and executed on virtually any digital infrastructure, including legacy systems. Thanks to an *ex ante* systematic human review, CATALA-created software will also comply with data protection law (§4). Finally, we include a technical overview of the language (§5) and explain how it meets the requirements.

This initiative belongs to the broader "rules as code" movement. However, we deliberately omit from this article considerations on co-designing legal statutes and computer programs. Rather, we take existing statutes for granted. Exploring the effectiveness of CATALA as a co-design tool is left for future work.

2 Existing Algorithmic Implementations of Statutes

Drawing from US and French Law, this section will provide an overview of usages and failures of algorithmic implementation of statutes.

2.1 Examples in US and French Law

Although there has been a boom in studies on the use of artificial intelligence tools by governments, algorithmic tools implemented in legal expert systems such as models or scoring systems have been relatively neglected, as Coglianese and Ben Dor (2019) report. However, these tools are widely used by governments and private companies to calculate social benefits and tax. Two examples, one drawn from US law and the other from French law, will show how public and private entities use algorithms for administrative purposes.

In the US, taxes are collected by the Federal government as well as by individual States. It is the responsibility of the taxpayer to compute for themselves the correct amount of taxes that they owe, depending on their income. Because of the complexity of this task, private companies have developed legal expert systems in order to help individuals fill out their tax forms. The most widely used program is *TurboTax*, as cited by Ruhl and Katz (2015). According to Lawsky (2020), there is reason to believe that this software is based on the tax forms prepared by the government. Hence, it is on the basis of these forms, themselves based on the US Tax Code, that an individual's tax is computed. The IRS (US Tax Agency) also operates an internal legal expert system to check tax returns once they are submitted[1].

In France, taxes are also computed by a legal expert system. Unlike in the US, though, the French government itself develops and maintains the software which computes the amount of taxes due. Led by the Directorate of Public Finances (DGFiP), some fragments of this software have been rendered public and are available online. Social benefits algorithms, however, are not widely published, though they can theoretically be accessed upon request.

Even if the size and responsibilities of the public sector for computing taxes is greater in France than in the US, some tax-related algorithms are commonly operated by the private sector. This is the case of private sector employees' payroll taxes and contributions to Social Security, computed by closed-source legal expert systems of companies such as ADP or PayFit. In both the US and France, software is used as a means to simplify tax collection and the distribution of social benefits. But as we will see next, current legal expert systems have some major issues and may even fail to comply with the law. In the next section, we will use a French example to illustrate the consequences of bugs.

2.2 Legacy Code and Industrial Failures: the French Example

The legal expert systems responsible for tax computation in large organizations sometimes correspond to what is called "legacy code". Legacy code is a term coined by the software industry to designate large, complex systems whose lifespan has exceeded the tenure of its original programmers[2]. According to scholars, legal expert systems used in large financial audit firms were created around 1990, as reported by Waterman et al. (1986); Bell (1985); Brown and Murphy (1990), and are still in use today in at least some private companies[3]. For the public sector, the French systems for income tax described by Merigoux et al. (2020) and the family benefits computation[4], were both created in 1990. The IRS system is even older and it is still operating 1950-era software and hardware, in spite of the multiple failed modernization attempts reported by Whitmore et al. (2008).

[1] Source: Internal Revenue Manual.

[2] For a more comprehensive description, see Victor (2013).

[3] Source: representatives from Ernst & Young, PwC, Deloitte and KPMG at the "Machine Intelligence and the Future of Professional Tax Services" panel during the 2020 UC Irvine Tax Symposium.

[4] Source: French Commission for Accessing Administrative Documents, notice n°20181891, 2019

Legal expert systems quickly suffered from the discontentment of users because of their poor usability, according to Leith (2016). After more than 30 years of existence, they have now acquired all the general characteristics of legacy code: use of obsolete technologies no longer taught in university courses, loss of expertise on critical portions of the source code, as original programmers retire, and a "plaster on a wooden-leg" approach to modification and maintenance.

These characteristics jeopardize the ability of the system to be adapted to new functional requirements. For example, in the case of systems implementing a statute, any modification of the statute entails a corresponding modification of the software. But because of the complexity and the fast-paced nature of tax law reforms, updates have to be frequent. As a consequence, maintenance of the systems becomes very difficult and costly.

The traditional solution to legacy code is migration. Migration, as described by Ganesan and Chithralekha (2016), boils down to creating a new system from scratch using modern technologies. Gaie (2020) emphasizes that during the migration, both systems have to run in parallel to ensure that they reach the same results[5]. However, in France, several migration attempts have led to a number of high-profile industrial failures.

More precisely, two migrations for public-sector legal expert systems have had catastrophic consequences for their users : the Louvois[6] army payroll computation system and the CIPAV[7] "auto-entrepreneur" pension rights computation system. In both cases, the State contracted a private company to undertake this task, but it failed to implement the legal specifications correctly. After years of maintenance and bug-fixing, these two migrated systems were still producing unreliable outputs, requiring extensive human supervision. This suggests that the complexity of the legal landscape has increased since 1990, as hinted by Cottin (2006). So much so that implementing correctly a legal expert system from scratch seems to be now out of reach from 1990-era traditional development methods described by Laplante and Neill (2004), still widely used in the software industry.

As these examples show, the current situation of legal expert systems is paradoxical. Even though their use is pervasive and critical in many large public or private organizations, they have become legacy code increasingly hostile to maintenance and migration. While it seems that there has not been any recent survey about the correctness of legal expert systems, the facts presented above should cast serious doubts on whether these systems are functionally equivalent to their reference statutes. And, as we will show in the following part of the article, bugs have serious legal and democratic implications.

3 Issues in Algorithmic Implementations of Statutes

Incorrect implementations of statutes have legal and democratic implications. Some solutions exist but they fail to address all of the implications.

[5] Any disagreement should correspond to a bug in the old system.

[6] Source: France Inter, Jan. 2018

[7] Source: France Inter, Jan. 2016. Hundreds of subscribers were threatened to lose their pension rights.

3.1 Legal and Democratic Implications

The shortcomings of legal expert systems have two important consequences in the context of French and EU law. First, bugs which can lead to miscalculations, and these miscalculations might be illegal. Second, because of the enormous scale of tax collection, it is impossible for an individual to systematically review the results generated by algorithmic tools. Such a barrier to human intervention may breach data protection law.

To begin with miscalculations: bugs in legal expert systems used by governments or private companies can lead to errors in the computation of taxes and social benefits. In 2009, for example, a French retirement scheme miscalculated pensions. For several years, the program failed to provide the correct amount of money to its beneficiaries[8]. Because these errors are sensitive matters, governments rarely acknowledge such bugs. As a consequence, literature on the subject is difficult to find. Nonetheless, miscalculation of taxes and social benefits might have important legal consequences. To understand these consequences, a distinction between algorithms that implement statutes and other kinds of algorithms must be drawn. Put simply, algorithms that implement statutes must respect the statute they implement. In other words, they are not independent from the text. Algorithms that compute taxes fall into this category. On the contrary, when an algorithm does not implement a statute, computer scientists have much more flexibility. Although they have to be in compliance with general rules, the algorithm does not have to be strictly equal to a statute.

This distinction is simple, but its consequences are not. Indeed, under French and EU law, there are no legal rules that oblige statute-implementing algorithms to strictly comply with the statute they implement. In other words, it could be difficult to find a legal basis through which an algorithm that does not comply with the statute it implements could be contested. This is true for French law, and for many other countries as well. But an interesting legal ground could be used as a substitute. Under French law, public algorithms can be defined as "actes administratifs", as noted by Huttner and Merigoux (2021). "Actes administratifs", under French public law, are decisions taken by the administration. As such, any act that falls under the category "acte administratif" can be contested in front of a judge. If an "acte administratif" is in breach of the law, it can be annulled. If we apply this logic to public algorithms, it means that an algorithm in breach of a statute can be annulled. The miscalculations resulting from a bug in the implementation are a breach of a statute. As such, there is a possibility for citizens to challenge the legality of the algorithm. If the challenge is successful, then the algorithm can be annulled and all the decisions taken on its ground – meaning amount of taxes paid by the individual – can be annulled, too. This is, of course, a serious democratic issue.

Secondly, if the results of the legal expert system are not reviewed by a human, they might be in breach of International and European Law. Indeed, this would fall under article 9 of the Convention 108+ of the Council of Europe for the protection of individuals with regard to the processing of personal data, which gives every individual the right not to be subject to a significant decision based solely on automated

[8] Source: press release of the French national pension agency (CNAV), May 13th, 2009.

processing of data. Similar protection can be found under article 22 of the GDPR. Article 22 of the GDPR states that individuals have the right not to be subject to a decision based solely on automated processing which produces legal effects. While there are discussions on the exact meaning of article 22, there is a consensus that it refers to decisions excluding human involvement. In other words, an automated decision happens when no human reviews the results[9]. There are exceptions to this right. For example, Member States can grant specific exemptions. Nevertheless, even when such exemptions are applicable, suitable measures must be put in place to safeguard the data subject's rights, freedoms, and legitimate interests. In France, for instance, such an exemption was put in place in 2018 for administrative purposes. As a result, the government is legally permitted to use automated tools to make decisions, including the computation of taxes[10]. But since legal expert systems are created to automate tax computation, there is, by definition, no human involvement and no human verification of the results. This means that, if these algorithms malfunction or lack suitable safeguards against negative impacts on individual rights, freedoms, and legitimate interests, they might be in breach of article 22 of the GDPR. Breaching the GDPR has severe consequences. Infringements of article 22 of the GDPR can lead to administrative fines up to 20 million euros[11].

Two final elements can be added. Under French law, public algorithms are subjected to an obligation of control by the administration ("maitrise de l'algorithme"). This obligation can be found under article 47 of the French Data Protection Law[12]. The article can be translated as follows : "the controller ensures the control of the algorithm and its evolution in order to explain to the individual the way the algorithm was implemented"[13]. It means that the administration must understand how the algorithm works and how its results are obtained. As a consequence, this article might be used as another ground to challenge public algorithms. This obligation could be a source of inspiration for other countries, opening grounds to contest public algorithms.

A second source of inspiration can be found in the Canadian Directive on Automated Decision-Making, which took effect on April, 1, 2019. The aim of the Directive is to ensure that 'Automated Decision Systems are deployed in a manner that reduces risks to Canadians and federal institutions, and leads to more efficient, accurate, consistent, and interpretable decisions made pursuant to Canadian law[14]. Under

[9] Article 29 Data Protection Working Party. Guidelines on Automated individual decision-making and Profiling for the purposes of Regulation 2016/679, 3 October 2017.

[10] Article 21, Loi n°2018-493 du 20 juin 2018 relative à la protection des données personnelles.

[11] Art. 83 (5) (b), GDPR : "Infringements of the following provisions shall, in accordance with paragraph 2, be subject to administrative fines up to 20 000 000 EUR, or in the case of an undertaking, up to 4 % of the total worldwide annual turnover of the preceding financial year, whichever is higher : (...) (b) the data subjects' rights pursuant to Articles 12 to 22".

[12] Loi n° 78-17 du 6 janvier 1978 relative à l'informatique, aux fichiers et aux libertés, dite « informatique et libertés », telle que modifiée par la loi n° 2018-493 du 20 juin 2018 relative à la protection des données personnelles.

[13] Art. 47 (2) (2), Loi informatique et libertés : « le responsable de traitement s'assure de la maîtrise du traitement algorithmique et de ses évolutions afin de pouvoir expliquer, en détail et sous une forme intelligible, à la personne concernée la manière dont le traitement a été mis en œuvre à son égard ».

[14] Art. 4 (1), Directive on Automated Decision-Making.

the Directive, algorithms used to recommend or make an administrative decision must be subject to an impact assessment. They must also be tested for unintended data biases and other factors that may unfairly impact the outcomes[15]. These rules could also provide new ways of contesting public algorithms.

Overall, while there are already legal grounds which can be used to contest faulty public algorithms, specific new measures taken under French, EU, and Canadian law demonstrate the wide and increasing severity of these issues. At the same time, they prove that there is a growing international appetite to address such problems.

3.2 Problems with Existing Solutions

There are several existing solutions to prevent the legal and democratic consequences of bugs. First, preventing bugs from happening in the first place through case-based testing. Second, improving transparency of algorithms and make them more accessible for maintenance. The two solutions are difficult to implement, as we will see, because of a host of political and technical difficulties.

3.2.1 First Solution : Case Based Testing

Let's begin with case-based testing. Like any piece of software, legal expert systems programmers take measures to locate and fix bugs. Finding a bug in a legal expert system requires interaction between lawyers and programmers. The process is the following[16]. First, lawyers create a virtual test case or pick one from production data (*e.g.* a household fiscal data). This will constitute the set of test cases. Afterwards, they manually compute the expected output of the legal expert system based on the reference statute. They then compare the output of the system with the lawyer's expected output. In the case of a disagreement, lawyers and programmers discuss how they got to their results. When the discrepancy is located, the software is updated to output the correct result. The most important phase of the process is the discussion between programmers and lawyers. This is the phase where the legal requirements, with all their subtleties and varying interpretations emphasized by Lawsky (2017), are confronted to unambiguous computer code.

While this process is effective in improving the quality of the legal expert system, it suffers from the same limitations software testing is generally subject to. Indeed, quoting Dijkstra (1972), program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence. But most importantly, testing efficiency is tied to the quality of the set of test cases used. For testing to be efficient, the set of test cases has to be diverse and numerous. It is only in this condition that the integrality of the code can be checked for correctness. This poses various issues.

Firstly, in the case of legal expert systems, creating new tests is costly since it requires legal expertise. A legal expert system typically handles thousands of distinct

[15] Art. 6 (3) (1), Directive on Automated Decision-Making.

[16] Source: author's private discussions with public sector French legal expert systems programmers and publicly available beta.gouv.fr blog post (Feb. 2020)

situations. To reach full coverage, a test base should be able to deal with all of these situations. While it is difficult to get accurate and recent data on legal expert systems' test bases, the French income tax computation system uses around 800 tests (for approximately a hundred thousand lines of code)[17]. This suggests that legal expert systems are currently under-tested.

Moreover, the set of test cases is updated along with the software as the legal requirements change. This means that for each statute modification, each test of the set should be manually reviewed by a lawyer to determine whether it is affected or not by the modification. For this reason, keeping a complete and correct test set up to date is very costly.

Beyond mismanagement and poor software quality, it seems that there is currently no systemic incentive for large public or private organizations to maintain a diverse and numerous test set for their legal expert systems. If an individual contests the result of the algorithms, the organization simply provides for a manual human review of their case. In the best scenario, the case is turned into a test for the system[18]. But most of the time, it seems that the software is never updated and human agents have to manually correct the output in future occurrences of the buggy situation.

Overall, case-based testing is a good way to trigger interaction between programmers and lawyers, leading to a better legal expert system. But the high cost of maintenance for a good test set, coupled to low incentives for covering corner cases correctly, have *de facto* led to low levels of confidence for legal expert systems. Moreover, case-based testing does not help to achieve algorithmic transparency, especially if the test set contains production data that cannot be revealed.

3.2.2 Second Solution : Transparency

The second solution, algorithmic transparency, is often seen as a strong safeguard against issues arising from automated decision-making, as argued by Carlson (2017); Wachter et al. (2017b). As a consequence, European and French law have flirted with the idea of making such transparency a legal obligation. Under the GDPR, articles 13, 14 and 15 give every individual the right to access meaningful information about the logic involved in automated decision-making. While there has been debate about the meaning of this right, it is generally accepted that individuals have the right to ask for access to information on algorithmic processing, like Wachter et al. (2017a); Bayamlıoğlu (2017); Casey et al. (2019) point out. This, however, is far from a general transparency requirement. This holds true in France, as well. Administrations have to give access to meaningful information on the logic of administrative algorithms[19]. Every individual can ask to access these algorithms, under the supervision of an independent commission. As a consequence, the government created an online platform where source code can be found.

But the publication of algorithms is scarce and complicated. Merigoux et al. (2020) report that the source code for French income tax computation, for example,

[17] Source: information personally transmitted by the French Directorate of Public Finances (DGFiP) to the authors.

[18] This behavior can lead to privacy violations when the data is covered by tax secrecy for instance.

[19] Article L.311-3-1 of the French code on relations between the administration and individuals

was published in an incomplete form. Access to the source code for family benefits computations has been denied because "since the legal expert system is old and complex, the extraction of the source code for publication is not technically possible without disproportionate effort". As a testament to these difficulties, a French Member of Parliament was recently appointed to investigate the issues with administrative source code publication[20]. The US situation is better in that regard, since the IRS regularly publishes draft versions of its tax forms for public review.

4 CATALA as a New Solution for Algorithmic Implementations of Statutes

As we have seen, some current legal expert systems are outdated and suffer from a lack of confidence on the correctness of their results, as well as general opacity that conflicts with the growing demand for algorithmic transparency. In this section, we introduce a new production process for legal expert systems along with appropriate tooling that answers to the issues raised above. This production process is based on two complementary concept, which we believe can help solve these issues: formal methods and literate pair programming. After a short presentation of these methods, we will show how they can help resolve some issues relating to algorithmic implementation of statutes.

4.1 The Use of Formal Methods

Formal methods are a subdomain of computer science, sharing close ties with mathematics. Its premise, as laid out by the seminal paper by Howard (1980), is to consider computer programs as mathematical objects, on which theories can be applied and properties, such as correctness or safety, proven. Formal methods have been deployed in critical industrial sectors like avionics, or nuclear energy production, with tools such as Astrée by Blanchet et al. (2002) and FramaC by Cuoq et al. (2012). Indeed, these techniques are able to completely rule out entire classes of bugs from computer programs, including memory safety bugs or software crashes. In the case of avionics, the Astrée static analyzer was able to "prove completely automatically the absence of any runtime error in the primary flight control software of the Airbus A340 fly-by-wire system, a program of 132,000 lines of C"[21]. The French electricity producer EDF used Astrée, FramaC and other tools to detect potential bugs in the software controlling nuclear power plants, as reported by Ourghanlian (2015).

Another achievement of formal methods is the ability to prove functional equivalence between a program and its specification, assuming that both can be expressed using formalized languages, *i.e.* whose behavior is described precisely using mathematical terms. This ability is of high interest for legal expert systems. Given a formal specification of a legal statute, it is possible to use formal methods to produce an executable implementation that is guaranteed to behave in the exact same way. An

[20] Source: French decree of June 22th, 2020.

[21] `https://www.astree.ens.fr/`

analogous process has been used by Caspi et al. (1987) for software controlling critical industrial facilities in real time.

More generally, formal methods shift the discussion about correctness from individual test cases to the source code of the program itself, whose behavior is considered for all possible inputs. We believe that reference statutes have to be considered as the ultimate specification of legal expert systems. Hence, the correctness problem of legal expert systems boils down to agreeing on a reference formal specification of the statutes. Optimized and executable implementations should be derived from this specification using advanced compilation techniques like CompCert by Leroy (2006), that preserve the functional equivalence between the statute and the software that is produced from it.

Interestingly, formal methods have already been used to formalize part of statutes, as noted by Morris (2020). Consequently, the question raised sporadic academic interest, but has never led to large-scale deployments in the public sector. The most advanced project in that category is certainly the French-led OpenFisca[22]. OpenFisca now features a comprehensive legal expert system able to compute the amount of almost all French taxes and benefits. However, OpenFisca does not have a formal grounding and the system has yet to be deployed in a government agency responsible for the collection of taxes or distribution of benefits.

Building on formal methods, our solution also features literate pair programming.

4.2 The Use of Literate Pair Programming

While literate pair programming is a clear advantage for the implementation of statutes, it faces a particular challenge when it comes to law. This challenge originates from the special structure of law. One of the most important feature of CATALA is, consequently, to address this challenge.

4.2.1 Presentation and General Advantages

Our solution combines two software development processes: literate programming and pair programming. To begin with literate programming, as defined by Knuth (1984), the source code of a program is annotated line by line with a textual description of what the program is supposed to do. This a systematic approach to documentation, and a good fit for programs whose behavior is subtle or difficult to infer just by looking at the code. While literate programming has been around for a long time, its utilization in practice remains limited to niche domains where the need for safety and correctness mandate a very exhaustive documentation. For instance, the US military report by Moore and Payne (1996) describes a successful use as early as 1996; Nedialkov (2011) reports using literate programming for the implementation of a very rigorous numerical simulation tool. More recently, literate programming has known a more widespread adoption with the advent of the Jupyter tool, very popular as a communication medium between data scientists, as reported by Kery et al. (2018).

[22] `openfisca.org`

On the other hand, pair programming is part of the agile process of software development described by Beck et al. (2001). It consists of pairing two programmers when producing software. While one of the programmers is busy writing the code, the other programmer can think about more high-level aspects of the software, or catch bugs as they're being written.

Both literate and pair-programming are used as *ex ante* and systematic ways of increasing the quality of software. It is relevant to combine these two concepts and use them for legal expert systems. The combination of pair programming and literate programming is included in the method of *extreme literate programming*, coined among others by Pieterse et al. (2004). However, the older and broader software trend of extreme programming, as described by Beck (1998), already mentions literate programming as a good technique to use in complement of the extreme programming rules that include the following: "All production code is pair programmed".

First, let us examine the advantages of literate pair programming in the context of legal expert systems. We have shown that interaction between lawyers and programmers is crucial for debugging legal expert systems. During this interaction, both parties play a crucial and complementary role: lawyers ensure that the specification reflects lawful interpretations of the statutes, while programmers ensure that the specification is completely unambiguous, and can be turned into an executable program.

However, this systematic interaction cannot be achieved with traditional waterfalls or V-shaped software development processes. With such processes, lawyers produce first a verbose natural language specification document from the statutes. The programmers then translate this verbose specification document into code. We identity three pitfalls. First, the lawyers don't know whether their specification document is sufficiently unambiguous to be turned into code. Second, when confronted to ambiguous or imprecise specifications, the programmers make arbitrary decisions that may correspond to unlawful code. Third, there is no direct and systematic connection between any piece of the source code and the piece of the statute that justify it.

Literate pair programming solves all those pitfalls: a lawyer and a programmer can produce together (pair programming) the legal expert system by gradually annotating the law with code translation (literate programming). When each line of statutory text is annotated with a line of code that translates its meaning, lawyers and programmers can have a local discussion with a visual support about a specific legal requirement. This format should foster mutual understanding, and eventually build cross-competence for both the lawyer and the programmer. With this method, interdisciplinary interaction is systematic and *ex ante*, by contrast to case-based interaction which is limited and *ex post*. Using agile methods can also significantly decrease the cost of software production. The French portal for social benefits computation, mes-aides.gouv.fr, has been developed from scratch using agile methods for a total cost of 1.25 million euros (over 5 years). When its maintenance was transferred to a private company using more traditional development processes, the cost skyrocketed to at least 2 million euros annually[23].

Interaction between programers and domain experts is a common trope of critical software development, as described by Fischer et al. (2009); Bialy et al. (2017). This

[23] Source: beta.gouv.fr evaluation of the project.

phase of the software development can also be called “knowledge acquisition”, as in Kidd (1987), Cohn et al. (1988) or Hart (1992), in the context of software expert systems.

4.2.2 The Particular Challenge for Law

However, to use literate pair programming for legal expert system production, adequate tooling is needed. Indeed, the structure of statutes is not adapted to traditional literate programming. This is demonstrated by Lawsky (2018) in the case of the US Tax Code, but we've empirically observed that the results apply for French statutes, because of similarities in the drafting style. The crux of the issue is an antagonism between the structure of law and the structure of computer programs. Indeed, normal programming goes from the most special case to the most general case. Statutes do the opposite. Because of this, implementations of statutes need to resort to impractical encodings based on nested conditionals, obscuring the behavior of the code.

This main hurdle, coupled with the lack of existing tooling solving it, has led us to create a new programming language designed to enable statute literate programming: CATALA[24]. Equipped with this appropriate tool, we ideally envision for CATALA statutes specifications to be published as open-source software, complementing existing publications of legislative texts. Coupled with state-of-the-art, formalism grounded, open-source compilation and interpretation tooling, this new method of producing legal expert systems could help to solve both the correctness and explainability issues of current solutions while driving legal expert systems maintenance costs down. All these features are present in CATALA[25].

To evaluate our claims about the efficiency of pair literate programming with CATALA, we conducted a medium-sized case study by implementing the computation of the French family benefits[26]. The program, counting two thousand lines of code including both the verbatim legislative and regulatory sources, as well as the CATALA code, was completed in about five pair programming sessions of two hours each plus about five hours of software engineering work for fixing software errors and compiling to a ready-to-use Javascript library. The resulting Javascript library was then embedded into the CATALA website to power a simulator that computes the amount of family benefits on-the-fly depending on inputs provided by the user of the website. After writing the code as well as some test cases, we compared the results of our program with the official state-sponsored simulator `mes-droits-sociaux.gouv.fr`, and found no issue. However, the case where a child is in the custody of social services was absent from the official simulator, meaning we could not compare results for this case. Fortunately, the source code of the simulator is available as part of the OpenFisca software project described by Shulz (2019). The OpenFisca source file

[24] Pierre Catala is, together with Lucien Mehl, a pioneer of French legal informatics, having authored works such as Catala et al. (1974). Beware, the name CATALA is typographically close to the name of the Catalan language written in Catalan : Català. However, we believe that the very narrow scope of our programming language is not prone to set any confusion given the existing wide influence of the Catalan language and culture.

[25] `catala-lang.org`

[26] `https://catala-lang.org/en/examples/family-benefits`

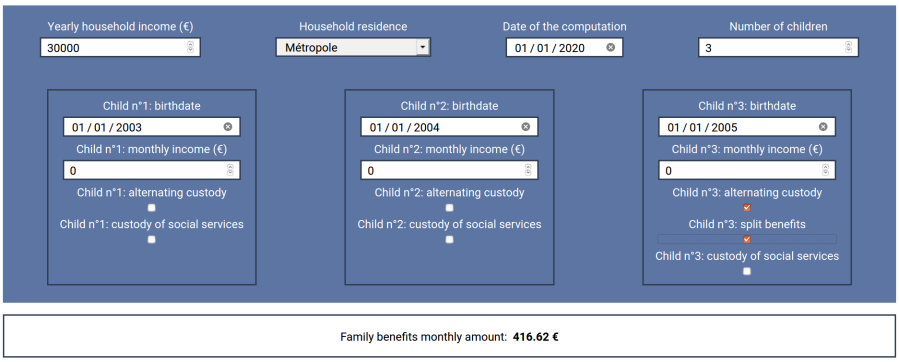

Fig. 1 Screenshot of the Web family benefits simulator powered by CATALA

corresponding to the family benefits, amounts to 365 lines of Python. After close inspection of the OpenFisca code, a discrepancy was located with the CATALA implementation. Indeed, according to article L755-12 of the Social Security Code, the income cap for the family benefits does not apply in overseas territories with single-child families. This subtlety was not taken into account by OpenFisca, and was fixed after its disclosure by the authors.

Given the complexity of the French family benefits computation that involves special rules for overseas territories, split custodies and progressively decreasing revenue thresholds, this early achievement is encouraging. We believe this motivates further investigation and evaluation of CATALA's efficiency with respect to related work.

5 The Technical Overview of CATALA

A complete technical description of the CATALA language and compilation ecosystem is presented in Merigoux et al. (2021). This section presents an overview whose goal is to be more accessible for scholars outside the subfield of formal methods.

5.1 The CATALA Frontend

To introduce the user-facing side of CATALA, we will illustrate the translation of a part of the US Internal Revenue Code into CATALA.

5.1.1 The Logical Structures of §121 of the US Internal Revenue Code

The following sentence is quoted from §121 of this Code, titled "Exclusion of gain from sale of principal residence":

> **(a) Exclusion**
> Gross income shall not include gain from the sale or exchange of property if, during the 5-year period ending on the date of the sale or exchange, such

property has been owned and used by the taxpayer as the taxpayer's principal residence for periods aggregating 2 years or more.

By applying the above ¶ (a), an US taxpayer can avoid paying taxes on the gains from the sale of his former principal residence. However, ¶ (b) of §121 immediately list some limitations to the general principle of ¶ (a):

(b) Limitations – (1) In general
The amount of gain excluded from gross income under subsection (a) with respect to any sale or exchange shall not exceed $250,000.

The first limitation caps the amount that can be excluded from the gain. It is an example of a common logical artifact of statutory drafting style, namely out-of-order definitions. Concretely, it means that the relevant rules for computing the value of a quantity – here the amount of gain excluded from gross income – are scattered in multiple paragraphs or sections of the statute. This feature is adversarial to literate programming, as traditional programming language do not allow more than one definition for a single variable. Continuing our exposition of §121, we enter ¶ (b)(2) and its first item (A):

(2) Special rules for joint returns
In the case of a husband and wife who make a joint return for the taxable year of the sale or exchange of the property—
(A) $500,000 Limitation for certain joint returns
Paragraph (1) shall be applied by substituting "$500,000" for "$250,000" if—
(i) either spouse meets the ownership requirements of subsection (a) with respect to such property;
(ii) both spouses meet the use requirements of subsection (a) with respect to such property; and
(iii) neither spouse is ineligible for the benefits of subsection (a) with respect to such property by reason of paragraph (3).

This paragraph displays a very common feature in statutory legal drafting: exceptions. More particularly, (b)(2)(A) is an exception to (b)(1) with respect to setting the cap of gross income exclusion. As pointed out by Sergot et al. (1986) and later by Lawsky (2018), exceptions are a form of non-monotonic reasoning: as you walk through the statute, information and rules does not always add up logically, instead amending or subtracting substance from previous items.

This kind of reasoning corresponds to a particular kind of logic called default logic, that has already been extensively studied in Reiter (1987) or Brewka and Eiter (2000). Default logic is adverse to traditional programming paradigms, because it requires manual and subtle encoding of the exception structure into first-order logic conditionals. Hence, it tends to obscure the source code of legal expert systems.

5.1.2 The CATALA *Formalization of §121 of the Internal Revenue Code*

Knowing the challenges posed by this section of the Internal Revenue Code, we now present a formalization of the paragraphs mentioned earlier using the CATALA programming language. The following will act as a hands-on introduction to the concepts

of CATALA. Concerning literate pair programming, CATALA has been designed with pair programming and lawyers' review in mind. As such, it enjoys syntax with natural language keywords that can be adapted to different countries. Right now, CATALA supports French and English inputs.

Before translating the computational rules corresponding to each paragraph, we begin by a very common trope of programming: description of the data structures. Indeed, our CATALA formalization is meant to be a program executable by a machine, that computes a result from input data. Therefore, we first declare that our program will operate on periods of time with a begin and end date, and personal data about the taxpayer's filing (periods of property ownership and usage as a principal residence):

```
declaration structure Period:
  data start content date
  data end content date

declaration structure PersonalData:
  data property_ownership content collection Period
  data property_usage_as_principal_residence content
    collection Period
```

Then, we can setup our first instance of a CATALA key concept: scopes. The concept of scope is powered by a very important notion in computer science: abstraction. Programming abstractions are tools that help the programmer take a high-level view on what the code they are writing is doing. The converse of abstraction is low-level, verbose description. When programming, one can for instance abstract two operations doing the same thing on a different subject. The result of this abstraction is a unique function, applied to two different arguments.

A scope in CATALA maps directly to the function abstraction. Each scope has a number of context variables that will be the input, outputs or intermediate stores of relevant values for the computation. Scopes have to be declared before being used, and the declaration contains the list of scope context variables. Here, we declare a scope for the §121 income exclusion amount for a single person:

```
declaration scope Section121SinglePerson:
  context gain_from_sale_or_exchange_of_property content money
  context personal content PersonalData
  context requirements_ownership_met condition
  context requirements_usage_met condition
  context requirements_met condition
  context amount_excluded_from_gross_income_uncapped
    content money
  context amount_excluded_from_gross_income content money
  context aggregate_periods_from_last_five_years content duration
    depends on collection Period
```

Context variables, as well as a structure's data items, are annotated with the type of their contents. These type annotations act as a safeguard for programming, since it

prevents the user from writing erroneous code that mixes up apples and oranges, computationally speaking. As a special case, conditions are context variable containing a boolean value (true or false) that is false by default; this models the legal concept of condition.

The data structures and scope declarations are what we call *metadata*: they are the explicitations of the data and computational items that are referred to implicitly by the statutory text. Going forward, we can build on those to formalize the computation rules themselves. First, we tackle the base case of ¶ (a):

```
scope Section121SinglePerson:
  rule requirements_ownership_met under condition
    aggregate_periods_from_last_five_years of
      personal.property_ownership >=^ 730 day
  consequence fulfilled

  rule requirements_usage_met under condition
    aggregate_periods_from_last_five_years of
      personal.property_usage_as_principal_residence
        >=^ 730 day
  consequence fulfilled

  rule requirements_met under condition
    requirements_ownership_met and requirements_usage_met
  consequence fulfilled

  definition amount_excluded_from_gross_income_uncapped equals
    if requirements_met
    then gain_from_sale_or_exchange_of_property
    else $0
```

The code above computes the number of days in the last five years during which the taxpayer has owned and used the property as a principal residence, and compare it against the 2-year threshold – 730 days according to Regulation 1.121-1(c)(1). The first three rules enable a context condition to be true under certain circumstances, while the last definition sets the value of a context variable in the base case.

Second, the formalization of ¶ (1)(b) will illustrate how Catala handles out-of-order definitions and exceptions:

```
scope Section121SinglePerson:
  definition gain_cap equals $250,000

  definition amount_excluded_from_gross_income equals
    amount_excluded_from_gross_income_uncapped

  exception definition under condition
     amount_excluded_from_gross_income_uncapped >=$ gain_cap
  consequence equals gain_cap
```

The first definition is straightforward. However, the definition of the amount excluded from gross income is split in two. In the base case, the definition refers to the quantity of ¶ (a). But there is an exception when the quantity of ¶ (a) exceeds the gain cap, in which case the amount excluded is capped. These two definitions would appear to be in conflict in a traditional programming language, but since the later one is labeled with "exception", CATALA knows that it should give precedence to it rather than the base case.

The last key CATALA feature to introduce it the notion of scope calling. Indeed, to be able to check whether a couple satisfies conditions (i), (ii) and (iii) of ¶ (b)(2)(A), we need to apply ¶ (a) on each spouse. This is an instance of computational modularity at play. The statutory text implicitly aims at reusing a part of its provisions as a modular subset, but setting different inputs. This feature aligns exactly with the concept of scope in CATALA. A particular scope can be reused and called inside a parent scope, acting like a regular function in the programmatic sense. To model (b)(2)(A), we define a new scope for a couple that calls the previous scope covering the computation for a single person:

```
49 declaration structure CoupleData:
50   data personal1 content PersonalData
51   data personal2 content PersonalData
52
53 declaration enumeration ReturnType:
54   -- SingleReturn content PersonalData
55   -- JointReturn content CoupleData
56
57 declaration scope Section121Return:
58   context return_data content ReturnType
59   context person1 scope Section121SinglePerson
60   context person2 scope Section121SinglePerson
61   ...
```

Equipped with the results of applying §121 for each spouse, we can write the rules that formalize (i), (ii) and (iii). Those rules will refer to the `person1` and `person2` variables, representing the two `Section121SinglePerson` scope calls' outputs. We will omit the corresponding code for brevity, but one can refer to the appendix of Merigoux et al. (2021) for the full implementation.

Closing this introduction to the user-facing side of CATALA, focused on §121 of the US Internal Revenue Code, we discuss in the next section the computer-facing side of the CATALA language.

5.2 The CATALA Backend

For a programming language, the backend designates the process through which the textual representation of the syntax (presented in the earlier section) is transformed into machine code. This process implies a number of consecutive transformations

forming a chain: the links are compilation steps, while the nodes are called "intermediate representations". Each intermediate representation contains a full description of the program at one stage of its transformation. This section describes how this compilation transformation chain is formalized, and how it is implemented.

5.2.1 The formalization of CATALA*'s compilation*

CATALA enjoys a design process guided by the best practices of programming languages research. More particularly, it has a formal semantics in the sense of Wright and Felleisen (1994), comprised of three complementary items: a description of the syntax of CATALA programs, a typing judgment that rules out ill-formed programs, and an operational semantics describing how valid programs execute.

To understand exactly what is the formalisation of CATALA, consider Figure 2 which features the whole compilation chain from the CATALA source code to existing programming languages like OCaml or Javascript. We explain briefly each compilation step :

- the first compilation step is parsing, which yields an abstract syntax tree of the surface language from the textual representation of the CATALA source code;
- then, the surface language undergoes "desugaring", a process by which redundant language features are eliminated in favor of a longer but simpler form;
- from the desugared language to the scope language, definitions and rules inside each scope are ordered according to the dependencies between them;
- scopes are then eliminated and transformed into regular functions of the default calculus, a representation that features a term inspired by the default calculus;
- this default calculus term is then eliminated and transformed to first-order logic conditionals that fit into the last intermediate representation, a very standard λ-calculus extended with exceptions and algebraic data types;
- this λ-calculus can then be easily translated to languages from the ML family such as OCaml, and to Javascript thanks to Vouillon and Balat (2014).

The λ-calculus is the simplest programming language, containing only functions. Invented by Church (1932), it has then been extended with various programming language features that bring more expressivity to programs. Its formalization is the gold standard of formal methods, and CATALA builds upon it.

The core semantic feature of CATALA is its use of the default logic of Reiter (1980) to enable defining a variable multiple times with preconditions. Multiple conditions can be triggered at the same time, for instance when the law defines two exceptional cases that overlap. In that case, the programmer has to specify (and justify by law) a priority between the conflicting exceptions. If no such priority is given, the execution will report an error to the user, indicating a black spot requiring legal interpretation.

This semantic feature is reflected as a special term in the default calculus intermediate representation. This term is given *ad hoc* typing and operational semantic rules, as well as a formal compilation scheme to λ-calculus using exceptions. The details of this formal endeavour, including a mechanized proof of correctness of the compilation of the default calculus term, can be found in Merigoux et al. (2021).

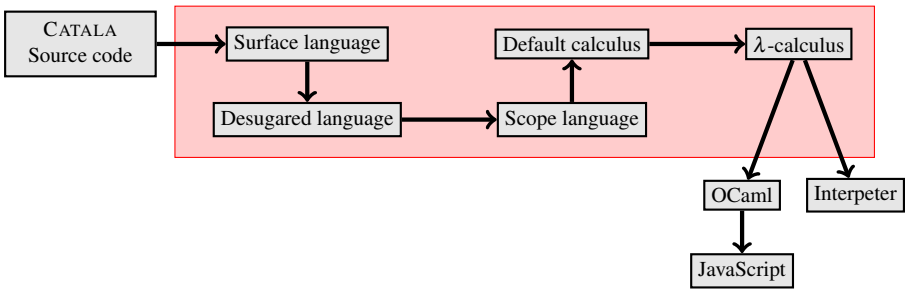

Fig. 2 High-level architecture of the CATALA compiler (red box)

Because of its formal grounding, existing off-the-shelf static analysers and automatic provers could be used on CATALA code. They could be used to check the coherence of the statute (no conflicts between exceptions) or whether the statute is valid with respect to requirements of another statute, higher in the hierarchy of statutes. For instance, article L521-1 of the French Social Security Code mandates that the amount of family benefits decreases with the household's income. The formula giving the amount, defined in several other articles, can be formally checked to satisfy this requirement.

5.2.2 The implementation of CATALA*'s compilation*

The CATALA compiler's implementation is written in the OCaml programming language and follows closely the description of the compilation chain presented in Figure 2, as well as the formal rules describing the semantics of each intermediate representation. The source code of the compiler is distributed under the Apache2 license and freely available on GitHub. As this is a relatively young research project, the interfaces and features of the compiler are still unstable and subject to changes and improvements.

The use of formal methods and compilation techniques to translate CATALA to other existing programming languages differ from related work coming from the logic programming and semantic web research communities. Starting with Sergot et al. (1986), several works have sought to encode pieces of legislation into Prolog, like the PROLEG framework by Satoh et al. (2010) or more recently some parts of the US Tax Code by Holzenberger et al. (2020). With the right extensions like Wielemaker et al. (2021), Prolog can function as a fully-featured rules engine on par with dedicated work like Flora2 by Yang et al. (2003). The leading LegalRuleML ontology-inspired markup language for legal texts, described in Athan et al. (2015), also uses logic programming as a semantic and interpretation backend through the RuleML architecture presented in Boley et al. (2010). All logic programming languages have in common the need for an interpreter or runtime system. While interpreted languages can be very efficient thanks to high-performance interpreters or just-in-time (JIT) compilers, they require software infrastructure that might not be available in the context of legacy production systems running on mainframes that

only supports proprietary and/or old versions of C, COBOL and Fortran compilers. Furthermore, with the advent of Web applications embarking legal expert systems, the possibility of running computations directly inside the Web browser using Javascript or WebAssembly would have the advantage of never sending any user data to a remote server, thus improving security and privacy. Of course, it is possible for logic programming languages to overcome all those limitations with some engineering tricks; but for CATALA, we opted for a simpler interoperability design.

By using advanced compilation techniques, CATALA programs could be compiled (translated) to any general-purpose programming language, including legacy languages like COBOL or Fortran. Indeed, adding a target language to the CATALA compiler is as easy as implementing a translation from a lambda calculus to the target language. To this date, the CATALA compile supports translations to Python, OCaml and Javascript, yielding ready-to-use source code libraries in those target languages. These libraries expose functions corresponding to the scopes defined in the CATALA program; these functions can then be called inside an existing IT system with data coming from a database or an input form. The CATALA compiler then guarantees that the result of calling these functions in the target language will be the same as calling the CATALA reference interpreter with the same data. This interoperability scheme is much more efficient, both in terms of program performance as well as development overhead. It also allows for separating the tax computing logic from the other parts of the system.

Last benefit of formal methods and compilation techniques: explainability requirements can also be addressed by a special compilation scheme that insert logging of the program's execution. Each log entry corresponds to a source code line, and therefore the statutory text provision that it annotates. This scheme addresses the needs for both individual and global algorithmic explainability, as long as the source code is open-source.

6 Conclusion

CATALA provides a solution for many of the problems addressed in this article. It reduces bugs and improves transparency in legal expert systems, and is, as such, in compliance with legal obligations. It also answers questions arising from changes in socio-economical context. To address the consequences of this change, organizations that operate legal expert systems should proactively seek to modernize their software infrastructure. As we have shown, it is likely that using traditional development processes will lead to industrial failures. But in the setting of an agile development process, CATALA would be the perfect tool with which to build correct and explainable new legal expert systems.

References

Athan T, Governatori G, Palmirani M, Paschke A, Wyner A (2015) Legalruleml: Design principles and foundations. In: Reasoning Web International Summer School, Springer, pp 151–188

Bayamlıoğlu E (2017) Transparency of automated decisions in the GDPR: an attempt for systemisation. International Data Privacy Law

Beck K (1998) Extreme programming: A humanistic discipline of software development. In: International Conference on Fundamental Approaches to Software Engineering, Springer, pp 1–6

Beck K, Beedle M, Van Bennekum A, Cockburn A, Cunningham W, Fowler M, Grenning J, Highsmith J, Hunt A, Jeffries R, et al. (2001) Manifesto for agile software development

Bell MZ (1985) Why expert systems fail. Journal of the Operational Research Society 36(7):613–619

Bialy M, Pantelic V, Jaskolka J, Schaap A, Patcas L, Lawford M, Wassyng A (2017) Software engineering for model-based development by domain experts. In: Handbook of System Safety and Security, Elsevier, pp 39–64

Blanchet B, Cousot P, Cousot R, Feret J, Mauborgne L, Miné A, Monniaux D, Rival X (2002) Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software. In: Mogensen, T, Schmidt, DA, Sudborough, IH (eds) The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones, Lecture Notes in Computer Science, vol 2566, Springer, pp 85–108

Boley H, Paschke A, Shafiq O (2010) Ruleml 1.0: the overarching specification of web rules. In: International Workshop on Rules and Rule Markup Languages for the Semantic Web, Springer, pp 162–178

Brewka G, Eiter T (2000) Prioritizing default logic. In: Intellectics and computational logic, Springer, pp 27–45

Brown C, Murphy D (1990) The use of auditing expert systems in public accounting. Journal of Information Systems pp 63–72

Carlson AM (2017) The need for transparency in the age of predictive sentencing algorithms. Iowa L Rev 103:303

Casey B, Farhangi A, Vogl R (2019) Thinking explainable machines: The GDPR's "right to explanation" debate and the rise of algorithmic audits in enterprise. Berkeley Technology Law Journal 34

Caspi P, Pilaud D, Halbwachs N, Plaice JA (1987) Lustre: A declarative language for programming synchronous systems. In: In 14th Symposium on Principles of Programming Languages (POPL'87). ACM

Catala P, Mehl L, Bertrand E (1974) Constitution et exploitation informatique d'un ensemble documentaire en droit (droit de l'urbanisme et de de la construction). Ed. du CNRS

Church A (1932) A set of postulates for the foundation of logic. Annals of mathematics pp 346–366

Coglianese C, Ben Dor L (2019) AI in adjudication and administration: A status report on governmental use of algorithmic tools in the United States. U of Penn Law School Public Law Research Paper(19-41)

Cohn LF, Harris RA, Bowlby W (1988) Knowledge acquisition for domain experts. Journal of computing in civil engineering 2(2):107–120

Cottin S (2006) Computer-assisted law-making-process: Information technologies and legal certainty – French experiments. In: International Conference, The State and the Legal System – Institutional Contemporary Transformations

Cuoq P, Kirchner F, Kosmatov N, Prevosto V, Signoles J, Yakobowski B (2012) Frama-c. In: Eleftherakis G, Hinchey M, Holcombe M (eds) Software Engineering and Formal Methods, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 233–247

Dijkstra EW (1972) The humble programmer. Communications of the ACM 15(10):859–866

Fischer G, Nakakoji K, Ye Y (2009) Metadesign: Guidelines for supporting domain experts in software development. IEEE software 26(5):37–44

Gaie C (2020) From secured legacy systems to interoperable services (the careful evolution of the french tax administration to provide new possibilities while ensuring the primary tax recovering objective). International Journal of Computational Systems Engineering 6(2):76–83

Ganesan AS, Chithralekha T (2016) A survey on survey of migration of legacy systems. In: Proceedings of the International Conference on Informatics and Analytics, Association for Computing Machinery, New York, NY, USA, ICIA-16

Hart A (1992) Knowledge acquisition for expert systems. McGraw-Hill, Inc.

Holzenberger N, Blair-Stanek A, Van Durme B (2020) A dataset for statutory reasoning in tax law entailment and question answering. arXiv preprint arXiv:200505257

Howard WA (1980) The formulae-as-types notion of construction. To HB Curry: essays on combinatory logic, lambda calculus and formalism 44:479–490

Huttner L, Merigoux D (2021) Traduire la loi en code grâce au langage de programmation Catala. Revue de droit fiscal (5):121

Kery MB, Radensky M, Arya M, John BE, Myers BA (2018) The story in the notebook: Exploratory data science using a literate programming tool. In: Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems, pp 1–11
Kidd AL (1987) Knowledge acquisition. In: Knowledge acquisition for expert systems, Springer, pp 1–16
Knuth DE (1984) Literate Programming. The Computer Journal 27(2):97–111
Laplante PA, Neill CJ (2004) The demise of the waterfall model is imminent. Queue 1(10):10–15
Lawsky SB (2017) Formalizing the Code. Tax Law Review 70(377)
Lawsky SB (2018) A Logic for Statutes. Florida Tax Review
Lawsky SB (2020) Form as formalization. Ohio State Technology Law Journal
Leith P (2016) The rise and fall of the legal expert system. International Review of Law, Computers & Technology 30(3):94–106
Leroy X (2006) Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. SIGPLAN Not 41(1):42–54
Merigoux D, Monat R, Gaie C (2020) Étude formelle de l'implémentation du code des impôts. In: 31ème Journées Francophones des Langages Applicatifs, Gruissan, France
Merigoux D, Chataing N, Protzenko J (2021) Catala: A Programming Language for the Law, URL `https://hal.inria.fr/hal-03159939`, working paper or preprint
Moore AP, Payne CN (1996) Increasing assurance with literate programming techniques. In: Proceedings of 11th Annual Conference on Computer Assurance. COMPASS'96, IEEE, pp 187–198
Morris J (2020) Spreadsheets for legal reasoning: The continued promise of declarative logic programming in law. Available at SSRN 3577239
Nedialkov NS (2011) Implementing a rigorous ode solver through literate programming. In: Modeling, Design, and Simulation of Systems with Uncertainties, Springer, pp 3–19
Ourghanlian A (2015) Evaluation of static analysis tools used to assess software important to nuclear power plant safety. Nuclear Engineering and Technology 47(2):212–218, DOI 10.1016/j.net.2014.12.009, URL `https://hal-edf.archives-ouvertes.fr/hal-01857446`
Pieterse V, Kourie DG, Boake A (2004) Literate programming to enhance agile methods. In: International Conference on Extreme Programming and Agile Processes in Software Engineering, Springer, pp 250–253
Reiter R (1980) A logic for default reasoning. Artificial Intelligence 13(1):81 – 132, special Issue on Non-Monotonic Logic
Reiter R (1987) Readings in nonmonotonic reasoning. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, chap A Logic for Default Reasoning, pp 68–93, URL `http://dl.acm.org/citation.cfm?id=42641.42646`
Ruhl J, Katz DM (2015) Measuring, monitoring, and managing legal complexity. Iowa L Rev 101:191
Satoh K, Asai K, Kogawa T, Kubota M, Nakamura M, Nishigai Y, Shirakawa K, Takano C (2010) Proleg: an implementation of the presupposed ultimate fact theory of japanese civil code by prolog technology. In: JSAI International Symposium on Artificial Intelligence, Springer, pp 153–164
Sergot MJ, Sadri F, Kowalski RA, Kriwaczek F, Hammond P, Cory HT (1986) The british nationality act as a logic program. Commun ACM 29(5):370–386
Shulz S (2019) Un logiciel libre pour lutter contre l'opacité du système sociofiscal. Revue francaise de science politique 69(5):845–868
Victor B (2013) Revisiting legacy systems and legacy modernization from the industrial perspective. Master's thesis, Universiteit Utrecht
Vouillon J, Balat V (2014) From bytecode to javascript: the js_of_ocaml compiler. Software: Practice and Experience 44(8):951–972
Wachter S, Mittelstadt B, Floridi L (2017a) Why a right to explanation of automated decision-making does not exist in the general data protection regulation. International Data Privacy Law 7(2):76–99
Wachter S, Mittelstadt B, Russell C (2017b) Counterfactual explanations without opening the black box: Automated decisions and the GDPR. Harv JL & Tech 31:841
Waterman DA, Paul J, Peterson M (1986) Expert systems for legal decision making. Expert Systems 3(4):212–226
Whitmore A, Rich E, Nelson MR (2008) Building on shifting sands: The structure of repetitive it project escalation, crisis, and de-escalation. In: Proceedings of the 26th International Conference of the System Dynamics Society, Athens, Greece
Wielemaker J, Arias J, Gupta G (2021) s (casp) for swi-prolog. In: 2021 International Conference on Logic Programming Workshops, ICLP Workshops 2021, CEUR-WS

Wright A, Felleisen M (1994) A syntactic approach to type soundness. Information and Computation 115(1):38 – 94

Yang G, Kifer M, Zhao C (2003) Flora-2: A rule-based knowledge representation and inference infrastructure for the semantic web. In: Meersman R, Tari Z, Schmidt DC (eds) On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 671–688