Programming Languages and Law

A Research Agenda

James Grimmelmann
james.grimmelmann@cornell.edu
Cornell University
Law School and Cornell Tech
New York City, NY, USA

ABSTRACT

If code is law, then the language of law is a programming language. Lawyers and legal scholars can learn about law by studying programming-language theory, and programming-language tools can be usefully applied to legal problems. This article surveys the history of research into programming languages and law and presents ten promising avenues for future efforts. Its goals are to explain how the combination of programming languages and law is distinctive within the broader field of computer science and law, and to demonstrate with concrete examples the remarkable power of programming-language concepts in this new domain.

CCS CONCEPTS

• **Software and its engineering → General programming languages**; **Domain specific languages**; • **Social and professional topics → Computing / technology policy**.

KEYWORDS

programming languages, law

ACM Reference Format:
James Grimmelmann. 2022. Programming Languages and Law: A Research Agenda. In *Proceedings of the 2022 Symposium on Computer Science and Law (CSLAW '22), November 1–2, 2022, Washington, DC, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3511265.3550447

1 INTRODUCTION

Computer science contains multitudes. It ranges from pure mathematics to quantum physics, from the heights of theory to the depths of systems engineering.

Some of its subfields speak to urgent problems law faces. Criminal procedure [60] and national security law [27] cannot regulate the world as it exists without taking account of whether, when, and how data can be kept private. Other subfields provide new perspectives on law. The "law as data" movement [9, 75] uses computational methods like topic modeling and decision-tree learning to analyze legal datasets in subjects as diverse as trademark infringement, [17] judicial rhetoric, [74] and the network structure of the United States Code. [59]

CSLAW '22, November 1–2, 2022, Washington, DC, USA

ACM ISBN 978-1-4503-9234-1/22/11.
https://doi.org/10.1145/3511265.3550447

I would like to argue that the computer-science field of *programming-language (PL) theory* speaks to law in both of these senses. Not only is it indispensable for answering certain kinds of practical legal questions, but its application can "illuminate the entire law." [36] Just as microeconomics provides a new and illuminating way to think about rights and remedies, and just as corpus linguistics provides a new and illuminating way to think about legal interpretation, PL theory provides new and illuminating ways to think about familiar issues from all across the law.

Consider, for example, the M++ project to formalize French tax law (described in more detail in Section 2.2). M++ is distinguished from the kind of routine systems engineering that tax authorities around the world perform on their computer systems by its rigorous use of PL theory to design a new programming language for describing the provisions of the French tax code. On the one hand, M++ is useful because it is a clean, modern language that is amenable to correctness proofs, improving the reliability of tax computations. On the other hand, M++ programs mirror the structure of the tax laws they formalize. Instead of treating the rules of tax law as an *ad hoc* design document, M++ treats the tax code *as though they were itself a program*, one meant to be "executed" by lawyers and accountants. The goal is not just to do the same thing as the tax code, but to do it in the same way, section by section, clause by clause.

To generalize, PL theory has something unique to offer law because there is a crucial similarity between lawyers and programmers: the way they use *words*. Computer science and law are both *linguistic* professions. Programmers and lawyers use language to create, manipulate, and interpret complex abstractions. A programmer who uses the right words in the right way makes a computer do something. A lawyer who uses the right words in the right way changes people's rights and obligations. There is a nearly exact analogy between the text of a program and the text of a law.

This parallel creates a unique opportunity for PL theory as a discipline to contribute to law. Some CS subfields, such as artificial intelligence (AI), deal with legal structures. Others, such as natural language processing (NLP), deal with legal language. But only PL theory provides a principled, systematic framework to analyze legal structures in terms of the linguistic expressions lawyers use to create them. PL abstractions have an unmatched *expressive* power in capturing the linguistic abstractions of law.

Over a decade ago, Paul Ohm proposed a new research agenda for "computer programming and law," describing in detail the value of executable code for legal scholarship: by gathering and analyzing information about the law more efficiently, by communicating

more effectively, and by writing programs to test and prove scholarly points. [87] More recently, Houman Shadab also called for legal scholars to improve their research by learning to code. [99] In one sense, this article takes up their call: PL theory can help legal scholars write better code. In another, it makes a more ambitious claim: if code is law, [72] then *understanding code is a way of understanding law, and vice versa*. PL theorists understand code in a profound and distinctive way; legal scholars understand law in a profound and distinctive way. They can and should learn from each other.

Indeed, they already are. Some parts of this article are forward-looking, describing the possibilities for what PL theory and law could become. But others are backward-looking, taking stock of what has already been accomplished. There is an active community of PL theory and law researchers, drawing on experts from both sides and building collaborations between the two. In January 2022, the ACM held the first Programming Languages and Law workshop (ProLaLa) as part of the annual ACM SIGPLAN Symposium on Principles of Programming Languages (POPL). Much of the work I will describe was presented there; indeed, I gave an earlier version of this article as a ProLaLa talk.

This article will review three case studies of the use of PL theory for law, and then present ten promising avenues of potential research. Some of them are being actively pursued, others are promising speculations. I hope to make the case for the value of bringing programming-language methods to bear on legal problems, and to provide some examples of such problems as an enticement to researchers.

2 CASE STUDIES

To understand what PL theory and law could do, we should start with what they have already done. Here are three examples of notable applications of programming-languages methods to law. The first – logic programming for law – shows how other advances in CS and law often rest on a foundation of PL theory. The second – a programming language for the French tax code – shows how a PL theory approach improves legal technology practice. And the third – a programming language for a portion of property law – shows how PL methods are a natural fit for many legal problems.

2.1 Legal Logic Programming Languages

There is an extensive subfield of AI and law. [9, 19, 34, 35, 38, 50, 75, 80, 89, 93, 98, 120] Its research program goes back nearly half a century, and it has its own dedicated association (the International Association for Artificial Intelligence and Law), conference series (the International Conference on Artificial Intelligence and Law), and journal (*Artificial Intelligence and Law*). One of the major lines of research in AI and law is *legal logic programing*: representing formal deductive legal reasoning using propositions expressed in a logic-programming language. [22, 53, 80, 98]

As researchers have discovered, however, general-purpose logic programming languages do not always capture the distinctive characteristics of legal reasoning. For example, many legal conclusions are *defeasible*: they are valid on the basis of present knowledge, but could fail if a known exception turns out to be the case. [67] But classical logic is monotonic: once a conclusion is established, it can never be falsified by additional hypotheses. For this reason, it can be hard to model legal reasoning in languages like Prolog whose semantics are monotonic. Capturing defeasible reasoning (for example in statute definitions with exceptions [71]) can require cumbersome circumlocutions.

Thus, some researchers have developed their own legal programming languages by adding defeasibility as a core language feature. Catala provides a clean and rigorous semantics for defeasible reasoning, which it uses to model statutory definitions with exceptions. [83] PROLEG implements default reasoning (in which conclusions have a default value even in the absence of evidence), which it uses to reason about issues governed by burdens of proof. [96]

Other legal logic programming languages add different features. s(LAW) models legal ambiguity and administrative discretion by generating multiple answers, depending on the resolution of relevant ambiguities. [8] LLD (a "Language for Legal Discourse") is a generic logic programming language enriched with temporal and deontic operators for reasoning about actions and obligations. [81, 82]

One could regard these examples merely as attempts to solve problems in AI and law. But I think it is telling that in so many cases the best way to solve a problem in AI and law has been to *create a programming language*. It is a sign that there are aspects of legal reasoning that programming languages are uniquely well-suited to capture.

2.2 The French Tax Code

The French tax code consists of about 3,500 pages of text, which the French tax authority (the DGFiP) must translate into an amount due for each taxpayer. Like any other large modern bureaucracy with millions of tasks, it does so by means of a computer program. But this program "relies on a legacy custom language [M] and compiler originally designed in 1990, which unlike French wine, did not age well with time." [84] The system is fragile and hard to maintain, and may contain bugs and mistakes.

A group of researchers are working with the DGFiP to transition its tax-computation system to a modern programming language backed up by a modern toolchain. They have reverse engineered M and given it a formal semantics, developed a new language (M++) with better formal properties and cleaner syntax, written a compiler (MLang) that supports both M and M++, and are using it to speed up, audit, debug, and ultimately *prove correct* the algorithmic implementation of the tax laws.[84] The following is a short fragment of a tax computation in M++ taken from their work:

```
compute_benefits():
  exists(taxbenefit) or exists(deposit):
    V_INDTEO = 1
    V_CALCUL_NAPS = 1
    partition with taxbenefit:
      NAPSANSPENA, IAD11, INE, IRE, PREM8_11
           <- call_m() iad11 = cast(IAD11)
    ire = cast(IRE)
    ine = cast(INE)
    prem = cast(PREM8_11)
```

```
V_CALCUL_NAPS = 0
V_IAD11TEO = iad11
V_IRETEO = ire
V_INETEO = ine
PREM8_11 = prem
```

There are two levels of insight here. The surface level is that MLang will make tax computations more reliable. But the deeper level is that because M++ makes it possible to express French tax law in a new way, it also makes it possible to *understand*, *reason about*, and *debate* French tax law in new ways. This is a classic programming-languages story about replacing an *ad hoc* system with one that rests on a firm theoretical and engineering foundation. M++ has rigorously specified semantics and its entire toolchain is built to support these semantics.

2.3 Future Interests

Orlando is a programming language for future interests in real estate.[14, 49] Its surface syntax is a stylized subset of the constructs used by lawyers to write conveyances like "`O conveys to A for life, then to B so long as B does not marry`" that create and transfer property interests. An accompanying implementation, Littleton, compiles these quasi-natural-langauge "programs" to a tree-based data structure modeling the different interests in a piece of property (here there are three: A's life estate, B's remainder, and O's possibility of reverter). Orlando has an operational semantics that specifies how those interests change over time in response to events (e.g. "`A dies`"). And Littleton can reason about the interests in an Orlando tree to do things like correctly name interests and apply the Rule Against Perpetuities. Littleton is is not the only tool or heuristic for visualizing future interests, [7, 16, 91] but it uses the formal power of a programming-language approach to provide a clean and extensible system.

On the most basic level, Orlando is useful as a teaching tool for exploring the consequences of particular conveyances. A student can tweak a conveyance to see what difference a wording change makes, or try out different sequences of events to see how the state of title evolves, or even turn on and off doctrines like the Rule in Shelley's Case. A teacher can put examples up on the screen and walk through them step by step, instantly updating based on questions and hypotheticals.

But on a deeper level, Orlando's syntax and semantics *themselves* are are a scholarly claim about the underlying structure of the law of future interests. They are based on the rules codified in the Restatement (First) of Property [90] and glossed in generations of treatises and study aids. [20, 37, 88, 106, 121] They make precise, testable claims about property doctrines. Disagreements about property law can be reformulated as different rules in a formal semantics.

3 RESEARCH TOPICS

Let us turn to what PL theory has to offer to law – and what law has to offer to PL theory.

3.1 Legal Domain-Specific Languages

New programming languages can provide formal models of specific legal subfields.

While the most familiar programming languages, like Java and C, are general-purpose langauges that can be used for a wide range of tasks, a *domain-specific language* (DSL) is specialized to meet the challenges of a specific problem domain. [40, 118] For example, the R programming language for statistical analysis and data visualization includes operators for performing calculations on arrays of data, while the Ink programming language for interactive fiction includes features that select which scene the player will experience next.

"Law" as a whole is much too large for a DSL (at least for now), but many legal subfields are the right size for a DSL. PROLEG, LLD, and s(LAW) are programming languages for legal logic programming. M++ is a DSL for tax law. Orlando is a DSL for future interests in property law.

The areas of law most amenable to formalization with DSLs are those where legal texts already already partly program-like. They are distinguished by three features. First they use *rules rather than standards*, so that they are amenable to formalization at all. Second, their participants *highly value clarity*, so that formalization offers real benefits. And third, they have *recurring patterned structures*, so that programming-language tools capture their modular generativity. [49, 101]

Transactional private-law fields are the most obvious low-hanging fruit. It is not a coincidence that there are now numerous contract DSLs. [6, 11, 12, 39, 56] Will drafting, for example, often involves the creation of trusts with common structures that have to be plugged together to fit the details of different family situations. Similarly, IP licensing breaks up rights into fairly standard elements, but the precise combination varies extensively from deal to deal. And many parts of property law have an underlying doctrinal structure that is itself recurring and patterned; programming languages for subareas like title assurance and land-use planning would take advantage of that structure.

3.2 Hybrid Contracts

Programming languages for contracts can combine the advantages of legally enforceable human-readable contracts and technically enforceable machine-readable code.

One of the most promising domains for legal DSLs is contracts, in part because here is where the most work has already been done. There has been extensive research products to model the deontic, multi-party, and event-driven structure of contracts. [6, 10–12, 24] More recently, the rise of blockchain-based smart contracts has driven extensive interest in developing good PLs for expressing them. [1, 65, 76, 104]

This last convergence suggests that there may be substantial conceptual and practical payoffs to developing systems that seamlessly hybridize human-readable terms and machine-readable logic. Users of such a system could write contractual terms once and them compile them both to "legal code" for humans and "computer code" for automated execution. [76, 110] [73] Carla Reyes has proposed (and written code for) a form that produces both an executable smart contract that can be run on a blockchain and a

legally effective financing statement that can be filed with a state office.[92] This approach could help solve one of the most vexing problems in smart-contract drafting: what to do when the actual code of a smart contract and its human-readable summary diverge? Drafting the two together helps prevent such divergences from occurring in the first place.

Another advantage of hybrid contracts is that formalizing the formalizable parts of them would help prevent – or at least detect – bugs in their logic. A human might miss an inconsistency in the natural-language version of a payment schedule, but a computer running the right algorithm over a software-language version of the same schedule would have a chance at flagging the problem. In addition, it would become more possible to automate execution of *part* of a contract, with responsibility for its successful performance being handled collaboratively by computers and humans.

More broadly, having a good programming-language-driven approach to hybrid contracts would help enormously in solving the difficult doctrinal problems they create. [3, 29, 47] It would help lawyers and judges conceptualize and answer questions about the legal effects of executing the automated part, and about the technical effects of legal enforcement.

3.3 Orthogonal Legal Primitives

Which legal concepts are simple and pervasive enough to serve as building blocks for legal programming languages?

There is a striking underlying simplicity in attempts to model distinctive features of legal reasoning. Orlando compiles an enormous range of property conveyances like "to Alice for life, then to Bob" to a small number of operators. It boils down the *effects* of a conveyance (rather than the *language* of the conveyance itself) into (1) the *termination* of interests under specified conditions ("to Alice *for life*"), and (2) the *sequencing* of interests ("to Alice …*then* to Bob"). Similarly, although Catala has a rich syntax of scopes, contexts, and definitions to allow users to track the structure of statutes, its semantics are built around extending standard lambda calculus with a single new feature: exceptions. It is defaults done right for the lambda calculus.

Beneath each of these legal programming languages, then, is a legal calculus: "a language with well-defined syntax and semantics that has features whose main purpose is to model particular aspects of its [legal] target domain." [15] The crown jewel of legal calculi is the Haskell-based language used by Jones, Eber, and Seward in *Composing Contracts* to model financial option contracts. [56] It has combinators for fundamental contractual operations like allowing a party to choose one of two obligations to take on (`or`) and allowing a party to choose when to take on an obligation (`anytime`). Its minimal set of ten combinators is both powerful and elegant; they carve up option contracts along the joints.

There is something fruitful about the exercise of identifying a legal concept and isolating it as a specific language feature. In the programming-language world, a language's elegance is often characterized in terms of the degree to which it is built from *orthogonal primitives*. [66, 108] Each of these features should be "primitive" in that it cannot be further decomposed, and the features should be "orthogonal" to each other in that they have simple and easily predictable interactions.

A profitable line of research would be to identify additional legal primitives and to systematize their relationships to each other. Existing examples already discussed include the sequencing and termination of legal interests (as in Orlando), the conjunction and disjunction of legal relationships (as in *Composing Contracts*), defaults and exceptions (as in Catala and PROLEG), and legal obligations and entitlements (as in LLD). This last example has a particularly distinguished heritage in law, as Hohfeld's system of jural opposites and jural correlatives is in a sense an informal and very early legal calculus. [51, 52] It has been the foundation for numerous efforts in AI and law, [4] and the interlocking nature of claim-rights and duties is essential to a programming language for contracts of any complexity. Other legal concepts that might usefully be translated into programming-language features include the creation and combination of entities in business associations law, hierarchies of authority among sources of law, the movement of jurisdiction over a case among courts during litigation, counterfactuals and hypotheticals, and concepts of transfer and notice in commercial law.

Any such effort needs to answer similar questions. First and foremost is the quest for clean minimal formalizations. This can be a difficult enterprise. Jones, Eber, and Seward observe:

> Identifying the "right" primitive combinators is quite a challenge. For example, it was a breakthrough to identify and separate the two forms of choice `or` and `anytime`, and encapsulate those choices (and nothing else) in two combinators.

Second, there is orthogonality. Consider, for example, the problem of combining deontic operators with operators for transferring property. Do the transferor's obligations "run with the land" to a transferee? In law, the answer is sometimes yes and sometimes no. Thus, obligation and transfer are not fully orthogonal; any system with both sets of operators must also have mechanisms to deal with their interaction by marking which obligations do and do not follow an interest in property. This is a hard problem, and that is part of what makes it interesting.

Third, the possibility that many legal concepts can be realized as programming-language primitives implies that *different* legal languages will often have *common* features. Thus, perhaps it would be useful to build legal-specific back-end support into the compiler toolchain, not to support any particular legal programming language but instead to support a range of projects using different languages. At the moment, the most common way to implement a legal programming language is to compile it to an off-the-shelf back-end. Is there enough in common among different legal PLs to justify adding law-specific features (e.g. for defaults or obligations) to one of those off-the-shelf back ends? I do not know the answer, but the question seems worth asking.

3.4 Legal Drafting Languages

Legal drafting could be made clearer and less error-prone by incorporating concepts from PL theory.

Programming-language theory helps programmers write better programs by giving them languages with features that promote clear and correct code. Perhaps PL theory can do the same for law. While some legal programming languages are designed to model

legal relationships that already exist,PL concepts could also be useful in improving legal drafting, where clarity and correctness are already important goals. [78, 107]

For example, Sarah Lawsky observes that the scope of statutory definitions is often ambiguous. [70] To be sure, some legal language is deliberately vague (e.g., "within a reasonable time"), but the scope of a statutory definition should be wholly unambiguous: either the definition of "home equity indebtedness" under the Internal Revenue Code should include the 1 million limitation on "acquisition indebtedness" or it should not. But the way the definitions are drafted does not resolve the issue. Lawsky proposes that legislative drafters should formalize their statutory definitions using first-order predicate logic, and then use those formalizations to guide their legislative drafting to make explicit the intended scope of each definition.

In other words, Lawsky proposes using first-order logic as a kind of *programming language for legal drafting*. This language has a feature – explicit scoping – that promotes clear and correct legal code. It is actually impossible to draft a tax provision in first-order logic with ambiguous scope, because that provision will violate simple and easily checked syntactic requirements that the language imposes on programs written in it. And once the provision is drafted in a logical form, it can then be "compiled" to the native format of the legal system – natural language – through a straightforward process that hopefully will not introduce errors of this sort. While, other scholars have addressed the value of logical formalization for legal reasoning and legal drafting, [5] Lawsky's analysis focuses attention on the way in which this formalization benefits from an appropriate programming language.

The power of this example prompts the question: *what other programming-language features would promote clear and unambiguous drafting*? A few possible answers include:

Variables and binding The inverse problem of specifying the scope of use of a given definition is specifying which binding a given reference refers to. PL theory has a rich collection of concepts (e.g., lexical versus dynamic scope), algorithms (e.g., data-flow analysis), and programming techniques (e.g., marking variables as global) to manage this problem. Legal drafters have their own rules of thumb to manage the problem. For example, patent drafters write "a" when introducing a new claim element and to write "the" when referring to that element subsequently, to make clear the distinction between binding and reference.

Cross-referencing Statutes and contracts are liberally sprinkled with cross-references (e.g. "for the purposes of subparagraph (e), 'days' shall mean business days"). But cross-references are fragile. As texts are amended, drafters may fail to catch all of the references to a renumbered section, resulting in mistaken or nonsensical cross-references. This too is a problem that programmers contend with, and numerous programming-language features promote correct coordination between different parts of a program. Function declarations, module systems, preprocessor includes, and type checking all enable coordination while trying to prevent the kinds of mistakes that it can introduce.

Counterfactuals and hypotheticals Legal drafting is full of counterfactuals and hypotheticals. For example, a party may be obligated to take action x if failing to do so *would result in* some state of affairs y. This type for hypothetical reasoning is one of the basic skills taught in law school, and lawyers rely on it extensively. PL theory has tools for reasoning about possible executions of a program, such model checking. And some language features, such as reflection, enable programs to reason about themselves. A good legal-drafting programming language might include features for compactly stating counterfactuals and hypotheticals.

Substitution Lawyers who use search-and replace to amend legal documents are already familiar with the concept of substitution. Sometimes it is explicitly a feature of the documents themselves, as when "in the case of an Adverse Event, all deadlines hereunder shall be extended for ten days." On the computer-science side, substitution is absolutely fundamental to lambda-calculus based semantics. PL theory has both a rich mathematical theory of substitution and an extensive arsenal of techniques and tools to employ substitution in a principled way when constructing and reasoning about programs.

Abstraction From polymorphism to virtual base classes, from abstract data types to functors, PL theory as a field is characterized by the deployment of powerful abstractions that massively generalize common programming patterns. Legal drafting's embrace of abstraction has been much more tentative; lawyers find themselves writing the same legal code over and over with minute variations. Good PL techniques for managing systematic abstractions could help lawyers avoid the tedium and risk of error that come from doing a hundred times what could be done once.

3.5 Legal Design Patterns

Common patterns in legal drafting and design should be systematically isolated and described, much as common patterns in software design already are.

In software engineering, a *design pattern* is a general, reusable template for solutions to a commonly occurring class of design problems. [42] The problems are not standardized enough that one can use a literally identical solution: if that were the case, one could simply reuse an existing library. Nor are they so diverse that each requires a bespoke solution from scratch. Rather, design patterns are useful in the middle ground, where the same kinds of problems recur, but the details are a little different each time. If this strikes you as being a lot like legal work, you are not wrong.

The roots of design patterns lie in the work of the architectural theorist Christopher Alexander and his collaborators; their 1977 book *A Pattern Language* described patterns like "light on two sides of every room"and argued that they fit into an interlocking, mutually reinforcing structure. [2] These ideas were carried over into the software engineering literature by developers and designers who appreciated its connections to modularity and the way that object-oriented programming, in particular, could be used to implement patterns. [41, 42] There and in related fields, such as user interface design, they have built up rich libraries of design patterns.

[32, 111] There is a close connection between design patterns and PL theory, because languages are frequently created to make certain types of design patterns easy to employ.

There is a nascent literature on legal design patterns, which identifies some common patterns in legal drafting. [33, 44, 94] This literature draws both on the original architectural theory and on the idea of design patterns in software.

A more ambitious agenda would seek to craft pattern languages for specific legal areas. In a sense, this is what some of the the formbooks and drafting manuals used by legal practitioners are getting at: they break contracts, licenses, settlements, security interests, regulations, and many other types of legal documents down into semi-standard parts, explain what each of those parts can do, and show how to fit them together. But this process can be systematized and scaled up. What are the proxies, decorators, and singletons in complex corporate transactions? What is the complete pattern language of will drafting?

A promising example of this approach is the United Kingdom's Office of the Parliamentary Counsel's *Common Legislative Solutions*, which provides a catalog of recurring solutions in regulatory design.[30] It follows the *Pattern Language* format, giving a high-level description of each pattern (e.g. "establish a statutory corporation"), followed by a detailed list of sub-elements requiring attention (e.g., "Should the statutory corporation have the power to borrow money?") and examples of the pattern in action (e.g., the Oil and Gas Authority and the Scottish Land Commission). More like this, please.

3.6 Design Principles

Broad design principles from programming-language theory are recognizable throughout the law.

Design patterns are solutions to specific commonly occurring problems. At a higher level of abstraction are design *principles*: general features of good software design that promote reliable, maintainable, and efficient code.

A key example is *modularity*: the decomposition of a system into subsystems that are weakly coupled to each other. The theory of modularity is rooted in cybernetics, [100] but has become a central principle of computer system design. [13] It is often achieved with explicit programming-language features. Programming-language support for separate compilation modules, for example, is literally about modularity: it's right there in the name. Other examples include functions, abstract data types, and objects.

Modularity in particular has already been fruitfully applied to legal theory. Henry Smith, who holds a Ph.D. in linguistics and draws heavily on the cybernetic theory of modularity, has shown how the boundaries of property (both physical and intellectual) are often drawn in modular ways to manage the information costs of dealing with other people's rights in things. [85, 102, 103] He and other scholars have applied modularity theory to legislative drafting, contracts, and internet regulation. [21, 54, 55, 101, 119, 123]

Other examples of broad programming-language design principles that may be relevant for law include recursion, compsitionality, type-safety, and extensibility. Legal drafting already makes *ad hoc* use of all of these principles, and systematizing them could help drafters be more precise. Ideal contract terms are not just modular but compositional: combining them produces no unexpected interactions, and replacing one adverse-events clause with another, say, should affect other parts of the contract. The breakdown of a contract into different kinds of clauses (representations versus promises, for example) is about typing, and a type-safe drafting framework would actually prevent drafters from using one in place of the other by mistake.

Corporate law uses modularity in concentrating legal relationships within discrete units; Coase's *Nature of the Firm* is in essence an argument about the optimal size of modules.[28] Corporate law also makes heavy use of extensibility in providing a set of default classes for corporate forms that can be extended with custom legal logic. Commercial law's treatment of transferrable obligations is modular and recursive: the passage of warranties along a chain of transfers is a recursive solution to the trust problem of dealing with strangers. The Federal Rules of Evidence's open-ended delegation of authority to the federal courts to make privilege rules is an extensibility framework. Other examples await those who are willing to go in search of them.

3.7 An IDE for Lawyers

Lawyers could benefit from drafting tools as sophisticated as those that programmers enjoy.

There was a time when programming a computer meant punching holes in paper cards, flipping switches for every bit, or even hand-wiring the program into physical connections between components. But one of the great accomplishments of computer science has been the conversion of programming from an arduous and punishing task into a process for leveraging human creativity. The modern toolchain supporting programmers – including tools for code editing, compilation, version control, code analysis, debugging, testing, and analytics – is powerful and extensive.

At the center of the toolchain are integrated development environments (IDEs) like Visual Studio Code, xCode, Eclipse, and Sublime Text. Consider just a few ways an IDE supports effective programming. As the programmer types, their code is automatically color-coded to keep the program's structure clear. The IDE suggests sensible auto-completions based not just on a static dictionary but on the rest of the program's code. The programmer can click on any function or variable to get more information about it (e.g. its type) and jump to other places it is used. With another click, they can reformat their code to make it easier to read. They can browse past versions of a code block to see how it has changed over time, and save changes to a shared repository. They can launch the program, set a breakpoint, and step through the code line by line to see how it behaves and where it goes wrong.

Now compare the legal drafting toolchain. By far the most common IDE for lawyers is Microsoft Word, and the most common version-control system is saving drafts as files with names like `RMD Agreement 3.2 MH Final 2 USE THIS ONE`. If you are lucky, you might use a specialized XML-based tool.[95] If you are very unlucky, you are still stuck with WordPerfect. The state of the art for most lawyers is track changes and a handful of Word plugins to assist with tasks like formatting citations. Other parts of the

toolchain are evolving quickly, especially case and document management. But the IDE lags behind.

What might a true modern IDE for drafting structured legal documents look like?

- It would feature syntax highlighting to distinguish different kinds of text, and automatic formatting for the structural framework of documents.
- It would use linting, type-checking, and other static analyses to enforce important invariants (e.g. that every defined term actually have exactly one definition) and to find common errors.
- It would detect and flag law smells, i.e., "patterns in legal texts that pose threats to the comprehensibility and maintainability of the law." [31]
- It would integrate directly to version control systems with easy-to-use features like branches, diffs, merges, and pull requests.
- It would integrate into project management systems so that team members could file to-dos against specific blocks of text, and into bug-tracking systems so that they could close out bug reports with the patches that fix them.
- It would also be integrated into knowledge-management systems used by law firms, agencies, and other institutions tracking immense quantities of documents.
- It would perform unit tests against any changes to ensure that they don't break existing features, and carry out fuzz testing with possible sequences of events to ensure robustness against unexpected contingencies.
- It would provide interactive simulation and debugging tools like visualization and breakpoints so that drafters could test out what the instrument they were creating will actually do in various situations.

3.8 Jupyter Notebooks for Law

Interactive visual tools for experimenting with code could be as powerful in law as they are in programming.

A great advantage of programming-language approaches is that they often improve understanding. Subsymbolic AI is notorious for having an explainability problem, [58, 97, 109] but for the most part not so PL theory. A good compiler's report of a type conflict directs the programmer's attention to a comprehensible and fixable mistake in their assumptions. Again, it is PL theory's embrace of the *structure* of programs that provides a foundation for making that structure comprehensible.

Within law and PL theory, a number of projects aim to present legal structures in an especially human-comprehensible way. Some of them are teaching tools for students and professors in a classroom setting, others are modeling tools for practitioners in a real-world setting, a few are both. As Lawsky observes, the modern tax form is in a sense a synthesis of formal and informal approaches, but it should be possible to do much better. [68]

One way in which programming-language techniques can help is visualization: although lawyers are notoriously verbal thinkers, sometimes a good diagram is worth a thousand words. For example, Littleton displays future interests using a "railroad diagrams" library originally created to show how a programming language's syntax works; Shawn Bayern's future-interests tool uses branching arrows. [16, 49] The two kinds of visualization are complementary, both to each other and to the textual formulations used by lawyers and law students. The use of programming-language techniques provides a more principled foundation than the informal diagrams drawn by generations of Property teachers. [7]

Another great advantage of programming-language techniques is interactivity. Unlike a static tutorial with a finite pre-canned set of examples, an interactive PL-based system can adapt on the fly to the user's queries, allowing in-depth exploration and the ability to test one's understanding with variations. Littleton and Bayern's interpreter both allow users to try a conveyance, examine the output, vary its details, try it again, and so on, creating a tight feedback loop. [16, 49] Lawsky Practice Problems uses programming-language techniques to *generate* an unlimited number of different tax practice problems, with the names, numbers, and doctrinal classification varying each time. [69]

Another way of integrating programming languages and law to improve understanding is through literate legal drafting. Literate *programming* is a style of programming in which a natural-language explanation of a program's functionality is interleaved with and generates the code implementing that functionality. [64] This vision inspires several recent projects in PL theory and law. The Catala project is working on having lawyers and programmers collaborate in a kind of pair programming, with the legal specification and program logic tightly integrated. [43, 84, 122] Similarly, the Accord Project's Cicero templating system is designed to embed machine-executable logic inline in the language of contracts. [1]

An ambitious goal combining visualization, interactivity, and literate programming would be to create the legal equivalent of a Jupyter notebook. [57] Such a notebook would freely intermingle natural-language text addressed to humans with executable segments addressed to computers, allow users to visualize legal structures using PL concepts like abstract syntax trees and control-flow graphs, and tighten the feedback loop between writing legal text and seeing what it does to a matter of seconds or less.

3.9 The Law of Software

Programming-language theory can answer doctrinal questions about software by describing what programs are and how they work.

Programming is not a regulated profession like medicine. But programs are subject to law in many ways. Here are just a few of the many doctrinal questions whose answers depend in part on questions about which programming-language theorists have relevant expertise.

- The patentability of an invention turns on whether it is an "abstract idea," and if so, whether it adds something "significantly more"to the idea. [116] The inherent abstraction of software means that this two-step inquiry is in play in any case claiming software functionality.
- Also in patent law, infringement occurs when a defendant supplies "components" of an invention in the United States for "combination" abroad."The Supreme Court held that a CD-ROM of Microsoft Windows was not a "component" of

a patented invention, because only the abstract software on the CD-ROM, not the CD-ROM itself, was installed on computers to make an infringing combination. [115]
- The Copyright Act protects computer programs as literary works. But not all programming languages are straightforwardly textual; in visual programming languages such as Scratch, a programmer manipulates graphical elements in two dimensions. Even in more traditional programming environments like Apple's Xcode, developers frequently combine program text with visual interface designs.
- Copyright protection does not extend to elements of a program that are "dictated by efficiency," by "external factors" such as compatibility, or by "widely accepted programming practices." [113]
- The Supreme Court has held that it is fair use to copy an application programming interface (API). [117] Dividing a program into "interface" and "implementation" requires detailed engagement with the details of the language in which it is written, and determining what elements are required for compatibility also requires considering how that language is compiled or interpreted.
- The First Amendment covers the publication of software as a means of communicating ideas to other programmers. [114] But using software for its functional effects, for example to spy on one's spouse, is not automatically protected. Drawing the line requires a theory of what constitutes the expressive speech in software. [112]
- Similarly, there may be a right *not* to write software; Apple has argued that being required to write code to unlock an iPhone would violate the First Amendment right against compelled speech. This argument would obviously fail if Apple were required to open a physical vault, so again one needs a theory of what is expressive and what is functional about software.
- Smart contracts are intended to supplement or replace legal contracts. Courts have a rich theory of how to interpret human-readable legal contracts, and they will need a similarly rich theory of how to interpret machine-readable smart contracts. [3, 47, 61]
- The Computer Fraud and Abuse Act prohibits some uses of computers "without authorization." Under some circumstances, what counts as "authorization" is determined by the program itself. [45, 46] (For example, a website authorizes or prohibits access by comparing a user-entered password to the password stored in its user database.) Again, there are interpretive questions that one needs a theory of computer-program meaning to answer.

3.10 Philosophical Questions

Lawyers and programmers can learn from each other what it means to interpret a text.

Legal and programming-language scholars both study interpretation: how to understand what a text means. The kinds of texts they study are different, but as we have seen repeatedly, this shared focus on text and its meaning is a large part of what makes their collaboration so fruitful. Indeed, this shared focus on text is *itself* a topic of interest. A deeper understanding of interpretation is not just of practical use in applying programming-language methods to legal problems: it can help theorists in both fields understand their fields better.

As we have seen, many doctrinal questions in PL theory and law are at the boundary between computer science and philosophy: what is a program, how does it work, and what does it mean? For example, the claim that software consist entirely of mathematics and is therefore not properly patentable or copyrightable [63, 86] is grounded in programming-language theories of the semantics of programming languages. But this is not the only possible way to conceive of software, [79] and the field of PL theory and law has much to contribute in complicating, challenging, and fleshing out this picture.

The fundamental theoretical question of PL theory and law is *what is the difference between how people and computers interpret texts*? [48, 77] Answers to this question tell us both what law and programming languages can do (because there are relevant similarities) and what it cannot (because there are unbridgeable differences). It tells us about what is truly fundamental about legal interpretation, and what is a historical accident of the fact that it developed in a pre-computer age. It tells us what changes about a rule when it is formalized, and it tells us about what the Church-Turing thesis means for society.

The fundamental practical question of PL theory and law is *how (if at all) should the availability of computers change how legal texts are written and interpreted*? I have argued that some (not all) legal rules should be made more formal, and that the legal drafting process should be more like the software development process. One could go much further: delegating more of the drafting process to computers, and using algorithms to interpret and apply laws. [18, 25, 26, 105] Of course, just because something can be done, doesn't mean it should. [23, 62] But this is a question that the legal profession urgently needs to ask itself, and programming-language scholars and software developers are in a prime position to help think through the possibilities and their consequences.

4 CLOSING THOUGHTS

Building a field of law and programming languages will not be easy; interdisciplinary work never is. Beyond the obvious point that each field has its own rich set of concepts and immense literature, the disciplines differ in more subtle ways. They have different research methods:

They have different research methods: PL researchers mostly build new things in teams, while legal researchers mostly study existing things alone. They have different standards of rigor: to caricature just a little, a PL paper is done when the proofs are filled in, a law paper when the footnotes are filled in. They have different authorities: "because Congress said so" is a good answer in law, but not in PL And they have different writing styles: to a PL scholar, the typical law-review article is shockingly long and repeats far too much that everyone already knows, while to a legal scholar, the typical PL paper is shockingly short and omits essential background and context.

One particular challenge is that scholars can be overconfident amateurs outside of their own field. The phrase "law-office history"

is usually a pejorative: it implies that legal scholars do poor history because they lack the training, the patience, and the motivation to get history right on its own terms. Of course, the reverse is also true: microeconomists sometimes get a bad rap in the legal academy for bursting in to long-running debates like the Kool-Aid Man, supremely confident that their stylized models hold all the answers.

Some of the answers to these challenges are the same in programming languages and law as they are in any interdisciplinary project. Collaboration will be essential, and some of the most interesting research efforts in the space involve teams with lead researchers from both programming languages and law. In addition, dual training in law and CS, although rare, is immensely helpful in bridging the two fields. There is a slow but growing trickle of researchers with knowledge of both, who have an important role in training the next generation of law and programming-language researchers. [87, 99]

I wish to close where I started, by asking again the question, why these two fields? Why law and programming languages? What do they have in common that other pairs of fields do not?

The answer, as I have emphasized throughout, is that law and programming languages share a common focus on how professionals use precisely structured linguistic constructions to do things in the world. Law has good reason to be interested in many CS fields, from cryptography to machine learning, but within CS, it is programming languages that most shares law's linguistic focus. And programming-language techniques can usefully be applied to many problem domains, but law happens to be particularly rich in the kind of problems for which programming-linguistic solutions can be useful. Historians and sociologists have their own interesting and important problems, but by and large they are not the kind of problems that a programming language can help solve.

If there is a lesson to take from the history of law and programming languages's engagement, it is that principled methods are often superior to *ad hoc* ones. PL theory has a long and impressive history of developing concepts, languages, and tools to tame the chaos of coding. Law and legal tech, which are engaged in their own eternal struggle to clean up the Augean stables of law and life, should welcome all the help they can get. Formalizing a body of law well enough to turn it into a programming language forces you to understand it in a far deeper way – and it is that hard-earned insight that law and programming languages promises.

ACKNOWLEDGMENTS

This work was supported by NSF Award FMitF-2019313. My thanks to the anonymous reviewers, to the organizers of and participants in the ProLaLa 2022 workshop, where I delivered an earlier version of this article as a talk, and to Aislinn Black, Shrutarshi Basu, and Sarah Lawsky. This article may be freely reused under the terms of the Creative Commons Attribution 4.0 International license, https://creativecommons.org/licenses/by/4.0.

REFERENCES

[1] Accord Project [2022]. *Accord Project.* Accord Project. https://docs.accordproject.org

[2] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, Angel Shlomo, et al. 1977. *A Pattern Language: Towns, Buildings, Construction.* Vol. 2. Oxford University Press.

[3] Jason G. Allen. 2018. Wrapped and Stacked: 'Smart Contracts' and the Interaction of Natural and Formal Language. *Eur. Rev. Cont. L.* 14, 4 (2018), 307.

[4] Layman E. Allen. 1974. Formalizing Hohfeldian Analysis to Clarify the Multiple Senses to Legal Right: A Powerful Lens for the Electronic Age. *S. Cal. L. Rev.* 48 (1974), 428.

[5] Layman E. Allen and C. Rudy Engholm. 1977. Normalized Legal Drafting and the Query Method. *J. Legal Educ.* 29 (1977), 380.

[6] Jesper Andersen, Ebbe Elsborg, Fritz Henglein, Jakob Grue, and Christian Stefansen. 2006. Compositional Specification of Commercial Contracts. *Int'l J. on Software Tools for Tech. Transfer* 8 (2006), 485.

[7] Roger W. Andersen. 1995. Present and Future Interests: A Graphic Explanation. *Seattle U. L. Rev.* 19 (1995), 101.

[8] Joaquín Arias, Mar Moreno-Rebato, Jose A Rodriguez-García, and Sascha Ossowski. 2021. Modeling administrative discretion using goal-directed answer set programming. In *Conference of the Spanish Association for Artificial Intelligence*. Springer, 258–267.

[9] Kevin D. Ashley. 2017. *Artificial Intelligence and Legal Analytics: New Tools for Law Practice in the Digital Age.* Cambridge University Press.

[10] Shaun Azzopardi, Gordon J. Pace, and Fernando Schapachnik. 2018. On Observing Contracts: Deontic Contracts Meet Smart Contracts. In *Proc. 31st Int'l Conf. on Legal Knowledge & Info. Systems (JURIX 2018)*. IOS Press, 21.

[11] Shaun Azzopardi, Gordon J. Pace, Fernando Schapachnik, and Gerardo Schneider. 2016. Contract Automata. *Artificial Intelligence & L.* 24 (2016), 203.

[12] Patrick Bahr, Jost Berthold, and Martin Elsman. 2015. Certified Symbolic Management of Financial Multi-Party Contracts. In *Proc. 20th ACM SIGPLAN Int'l Conf. on Functional Programming*. Association for Computing Machinery, 315.

[13] Carliss Y. Baldwin and Kim B. Clark. 2000. *Design Rules: The Pwer of Modularity.* MIT Press.

[14] Shrutarshi Basu, Nate Foster, and James Grimmelmann. 2019. Property Conveyances as a Programming Language. In *Proceedings of the 2019 ACM SIGPLAN International Symp. on New Ideas New Paradigms & Reflections on Programming & Software (Onward!)*. Association for Computing Machinery, 128.

[15] Shrutarshi Basu, Anshuman Mohan, Nate Foster, and James Grimmelmann. 2022. Legal Calculi. In *Programming Languages and the Law (ProLaLa)*. Association for Computing Machinery.

[16] Shawn J. Bayern. 2010. A Formal System for Analyzing Conveyances of Property Under the Common Law. *JURIX* 23 (2010), 139.

[17] Barton Beebe. 2006. An empirical study of the multifactor tests for trademark infringement. *Calif. L. Rev.* 94 (2006), 1581.

[18] Omri Ben-Shahar and Ariel Porat. 2021. *Personalized Law: Different Rules for Different People.* Oxford University Press.

[19] J.M. Trevor Bench-Capon, Gwen O. Robinson, Tom W. Routen, and Marek J. Sergot. 1987. Logic Programming for Large Scale Applications in Law: A Formalisation of Supplementary Benefit Legislation. In *Proc. 1st Int'l Conf. on Artificial Intelligence & L.* Association for Computing Machinery, 190.

[20] Thomas F. Bergin and Paul G. Haskell. 1984. *Preface to Estates in Land and Future Interests* (2nd ed. ed.). Foundation Press.

[21] Thomas F. Blackwell. 2000. Finally Adding Method to Madness: Applying Principles of Object-Oriented Analysis and Design to Legislative Drafting. *NYU. J. Legis. & Pub. Pol'y* 3 (2000), 227.

[22] Marc A Borrelli. 1989. Prolog and the law: Using expert systems to perform legal analysis in the uk. *Software LJ* 3 (1989), 687.

[23] Dan L Burk. 2019. Algorithmic fair use. *U. Chi. L. Rev.* 86 (2019), 283.

[24] John J. Camilleri, Gabriele Paganelli, and Gerardo Schneider. 2014. A CNL for Contract-Oriented Diagrams. *Controlled Nat. Language* (2014), 135.

[25] Anthony J Casey and Anthony Niblett. 2016. Self-driving laws. *University of Toronto Law Journal* 66, 4 (2016), 429–442.

[26] Anthony J Casey and Anthony Niblett. 2017. Self-driving contracts. *J. Corp. L.* 43 (2017), 1.

[27] Robert Chesney. 2021. *Cybersecurity Law, Policy, and Institutions.* https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3547103

[28] Ronald Harry Coase. 1937. The Nature of the Firm. *Economica* 4, 16 (1937), 386–405.

[29] Shaanan Cohney and David A. Hoffman. 2020. Transactional Scripts in Contract Stacks. *Minn. L. Rev.* 105 (2020), 319.

[30] The Parliamentary Counsel. 2022. *Common Legislative Solutions: A Guide to Tackling Recurring Policy Issues in Legislation.* Technical Report. Cabinet Office. https://www.gov.uk/government/publications/common-legislative-solutions-a-guide-to-tackling-recurring-policy-issues-in-legislation

[31] Corinna Coupette, Dirk Hartung, Janis Beckedorf, Maximilian Böther, and Daniel Martin Katz. 2022. Law Smells. *Artificial Intelligence and Law* (2022), 1–34.

[32] Christian Crumlish and Erin Malone. 2009. *Designing social interfaces: Principles, patterns, and practices for improving the user experience.* " O'Reilly Media".

[33] Lawrence A. Cunningham. 2006. Language, Deals, and Standards: The Future of XML Contracts. *Wash. U. L. Rev.* 84 (2006), 313.

[34] Cary G. Debessonet and George R. Cross. 1986. An Artificial Intelligence Application in the Law: CCLIPS, A Computer Program that Processes Legal Information. *High Tech. L.J.* 1 (1986), 329.

[35] Phan Minh Dung and Giovanni Sartor. 2011. The Modular Logic of Private International Law. *Artificial Intelligence & L.* 19 (2011), 233.

[36] Frank H Easterbrook. 1996. Cyberspace and the Law of the Horse. *University of Chicago Legal Forum* 1996 (1996), 207.

[37] Linda Edwards. 2009. *Estates in Land and Future Interests: A Step-by-Step Guide* (3rd ed. ed.). Wolters Kluwer.

[38] John P .Finan. 1981. LAWGICAL: Jurisprudential and Logical Considerations. *Akron L. Rev.* 15 (1981), 675.

[39] Financial Domain-Specific Language Listing. [2022]. . http://www.dslfin.org/resources.html

[40] Martin Fowler. 2010. *Domain-Specific Languages.* Addison-Wesley.

[41] Richard P Gabriel. 1996. *Patterns of Software.* Oxford University Press.

[42] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns Elements of Reusable Object-Oriented Software.* Addison-Wesley.

[43] Jason Garber. 2020. *Practical pair programming.* A Book Apart.

[44] Erik F. Gerding. 2013. Contract as Pattern Language. *Wash. L. Rev.* 88 (2013), 1323.

[45] James Grimmelmann. 2013. Computer Crime Law Goes to the Casino. https://www.techpolicy.com/Grimmelmann_ComputerCrimeLawGoesToCasino.aspx

[46] James Grimmelmann. 2016. Consenting to Computer Use. *Geo. Wash. L. Rev.* 84 (2016), 1500.

[47] James Grimmelmann. 2019. All Smart Contracts Are Ambiguous. *J. L. & Innovation* 2 (2019), 1.

[48] James Grimmelmann. 2022. The Structure and Legal Interpretation of Computer Programs. *Journal of Cross-Disciplinary Research in Computer Science and Law* (2022), to appear.

[49] James Grimmelmann, Shrutarshi Basu, Nate Foster, Shan Parikh, and Ryan Richardson. 2022. A Programming Language for Estates and Future Interests. *Yale Journal of Law and Technology* 24 (2022), to appear.

[50] Richard S. Gruner. 1989. Sentencing Advisor: An Expert Computer System for Federal Sentencing Analyses. *Santa Clara Computer & High Tech. L.J.* 5 (1989), 51.

[51] Wesley Newcomb Hohfeld. 1913. Some Fundamental Legal Conceptions as Applied in Judicial Legal Reasoning. *Yale L.J.* 16 (1913), 28–59.

[52] Wesley Newcomb Hohfeld. 1917. Fundamental legal conceptions as applied in judicial reasoning. *The Yale Law Journal* 26, 8 (1917), 710–770.

[53] Nils Holzenberger, Andrew Blair-Stanek, and Benjamin Van Durme. 2020. A Dataset for Statutory Reasoning in Tax Law Entailment and Question Answering. In *Proc. 2020 Nat. Legal Language Processing (NLLP) Workshop.* Association for Computational Linguistics, 31.

[54] Cathy Hwang. 2015. Unbundled Bargains: Multi-Agreement Dealmaking in Complex Mergers and Acquisitions. *U. Pa. L. Rev.* 164 (2015), 1403.

[55] Cathy Hwang and Matthew Jennejohn. 2018. Deal Structure. *Nw. U. L. Rev.* 113 (2018), 279.

[56] Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. 2000. Composing contracts: an adventure in financial engineering (functional pearl). *ACM SIGPLAN Notices* 35, 9 (2000), 280–292.

[57] Project Jupyter [2022]. *Project Jupyter.* Project Jupyter. https://jupyter.org

[58] Margot E Kaminski. 2019. The right to explanation, explained. *Berkeley Tech. LJ* 34 (2019), 189.

[59] Daniel Martin Katz and Michael James Bommarito. 2014. Measuring the Complexity of the Law: The United States Code. *Artificial Intelligence & L.* 22, 4 (2014), 337.

[60] Orin S Kerr. 2018. *Computer Crime Law* (4 ed.). Thomson/West.

[61] Gregory Klass. 2022. How to Interpret a Vending Machine: Smart Contracts and Contract Law. (2022). https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4045711

[62] Gregory Klass. 2022. Tailoring ex Machina: Perspectives on Personalized Law. *U. Chi. L. Rev. Online* (2022).

[63] Ben Klemens. 2005. *Math You Can't Use: Patents, Copyright, and Software.* Brookings Institution Press.

[64] Donald Ervin Knuth. 1984. Literate Programming. *Computer J.* 27, 2 (1984), 97.

[65] Jan Ladleif and Mathias Weske. 2019. A Unifying Model of Legal Smart Contracts. In *Proc. Int'l Conf. on Conceptual Modeling.* 323.

[66] Peter J. Landin. 1966. The Next 700 Programming Languages. *Comm. ACM* 9, 3 (1966), 157.

[67] Sarah Lawsky. 2017. *Nonmonotonic Logic and Rule-Based Legal Reasoning.* Ph. D. Dissertation. University of California, Irvine. https://escholarship.org/uc/item/59j2j45w

[68] Sarah Lawsky. 2020. Form as Formalization. *Ohio St. Tech. L.J.* 16 (2020), 114.

[69] Sarah Lawsky. [2022]. Lawsky Practice Problems. https://www.lawskypracticeproblems.org

[70] Sarah B. Lawsky. 2016. Formalizing the Code. *Tax L. Revieew* 70 (2016), 377.

[71] Sarah B. Lawsky. 2017. A Logic for Statutes. *Fla. Tax Rev.* 21 (2017), 60.

[72] Lawrence Lessig. 1999. *Code: And Other Laws of Cyberspace.*

[73] Matt Levine. 2016. Blockchain Company's Smart Contracts Were Dumb. *Bloomberg Opinion* (2016). https://www.bloomberg.com/view/articles/2016-06-17/blockchaincompany-s-smart-contracts-were-dumb

[74] Michael A Livermore, Allen B Riddell, and Daniel N Rockmore. 2017. The Supreme Court and the judicial genre. *Ariz. L. Rev.* 59 (2017), 837.

[75] Michael A. Livermore and Daniel N. Rockmore. 2019. *Law as Data: Computation, Text, & the Future of Legal Analysis.* SFI Press.

[76] Megan Ma. 2020. Writing in Sign: Code as the Next Contract Language?

[77] Megan Ma, Dmitriy Podkopaev, Avalon Campbell-Cousins, and Adam Nicholas. 2020. Deconstructing legal text_Object oriented design in legal adjudication. *arXiv preprint arXiv:2009.06054* (2020).

[78] Megan Ma and Bryan Wilson. 2021. The Legislative Recipe: Syntax for Machine-Readable Legislation. *Nw. J. Tech. & Intell. Prop.* 19 (2021), 107.

[79] Donald MacKenzie. 2001. *Mechanizing Proof.* MIT Press.

[80] L. Thorne McCarty. 1976. Reflections on TAXMAN: An Experiment In Artificial Intelligence And Legal Reasoning. *Harv. L. Rev.* 90 (1976), 837.

[81] L Thorne McCarty. 1989. A language for legal discourse i. basic features. In *Proceedings of the 2nd international conference on Artificial intelligence and law.* Association for Computing Machinery, 180–189.

[82] L Thorne McCarty. 2022. Position Paper: LLD is All You Need. In *Programming Languages and the Law (ProLaLa).* Association for Computing Machinery.

[83] Denis Merigoux and Liane Huttner. 2020. Catala: Moving Towards the Future of Legal Expert Systems. (2020). https://hal.inria.fr/hal-02936606/document

[84] Denis Merigoux, Raphaël Monat, and Jonathan Protzenko. 2021. A Modern Compiler for the French Tax Code. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction* (Virtual, Republic of Korea) *(CC 2021).* Association for Computing Machinery, New York, NY, USA, 71–82. https://doi.org/10.1145/3446804.3446850

[85] Thomas W. Merrill and Henry E. Smith. 2000. Optimal Standarization in the Law of Property: the *Numerus Clausus* Principle. *Yale L.J.* 110 (2000), 1.

[86] Eben Moglen. 1999. Anarchism Triumphant: Free Software and the Death of Copyright. *First Monday* 4 (1999). https://firstmonday.org/ojs/index.php/fm/article/view/684/594

[87] Paul Ohm. 2009. Computer Programming and The Law: A New Research Agenda. *Vilanova L. Rev.* 54 (2009), 117.

[88] Richard R. Powell. 1949. *The Law of Real Property.* Matthew Bender & Company.

[89] Henry Prakken and Giovanni Sartor. 2015. Law and Logic: A Review From an Argumentation Perspective. *Artificial Intelligence* 227 (2015), 214.

[90] 1936. *Restatement (First) of Property.*

[91] Mark Reutlinger. 1994. When Words Fail Me: Diagramming The Rule Against Perpetuities. *Mo. L. Rev.* 59 (1994), 157.

[92] Carla L Reyes. 2021. Creating Cryptolaw for the Uniform Commercial Code. *Wash. & Lee L. Rev.* 78 (2021), 1521.

[93] Edwina L. Rissland. 1990. Artificial Intelligence and Law: Stepping Stones to a Model of Legal Reasoning. *Yale L.J.* 99, 8 (1990), 1957.

[94] Matthew Roach. 2016. Toward A New Language Of Legal Drafting. *J. High Tech. L.* 17 (2016), 43.

[95] Giovanni Sartor, Monica Palmirani, Enrico Francesconi, and Maria Angela Biasiotti (Eds.). 2011. *Legislative XML for the semantic web: principles, models, standards for document management.* Springer Science & Business Media.

[96] Ken Satoh, Kento Asai, Takamune Kogawa, Masahiro Kubota, Megumi Nakamura, Yoshiaki Nishigai, Kei Shirakawa, and Chiaki Takano. 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, 153–164.

[97] Andrew D Selbst and Solon Barocas. 2018. The intuitive appeal of explainable machines. *Fordham L. Rev.* 87 (2018), 1085.

[98] Marek J. Sergot, Fariba Sadri, Robert A. Kowalski, Frank Kriwaczek, Peter Hammond, and H. Terese Cory. 1986. The British Nationality Act as a Logic Program. *Comm. ACM* 29 (1986), 370.

[99] Houman B. Shadab. 2020. *Software is Scholarship.* Technical Report. MIT Computational Law Report. https://papers.ssrn.com/sol3/Papers.cfm?abstract_id=3632464

[100] Herbert A Simon. 1996. *The Sciences of the Artificial* (3rd ed. ed.). MIT Press.

[101] Henry E. Smith. 2006. ModularIty in Contracts: Boilerplate and Information Flow. *Mich. L. Rev.* 104 (2006), 1175.

[102] Henry E. Smith. 2011. *Standardization in Property Law.* Edward Elgar, 148.

[103] Henry E. Smith. 2012. Property as the Law of Things. *Harv. L. Rev.* 125 (2012), 1691.

[104] 2022. *Solidity Documentation* (release 0.8.12 ed.). https://docs.soliditylang.org

[105] Lawrence B Solum. 2014. Artificial meaning. *Wash. L. Rev.* 89 (2014), 69.

[106] John G. Sprankling. 2017. *Understanding Property Law* (4th ed. ed.). Carolina Academic Press.

[107] Tina L Stark. 2013. *Drafting contracts: How and why lawyers do what they do.* Wolters Kluwer.

[108] Guy L. Steele, Jr. 1998. Growing a Language. (1998). http://www.cs.virginia.edu/~evans/cs655/readings/steele.pdf

[109] Katherine J Strandburg. 2019. Rulemaking and inscrutable automated decision tools. *Columbia Law Review* 119, 7 (2019), 1851–1886.
[110] Harry Surden. 2012. Computable Contracts. *U.C. Davis L. Rev.* 46 (2012), 629.
[111] Jenifer Tidwell, Charles Brewer, and Aynne Valencia. 2020. *Designing interfaces: Patterns for effective interaction design* (3 ed.). "O'Reilly Media".
[112] Lee Tien. 2000. Publishing Software As A Speech Act. *Berk. Tech. L.J.* 15 (2000), 629.
[113] United States Court of Appeals for the Second Circuit. 1992. Computer Associates Intern., Inc. v. Altai, Inc., 982 F.2d 693.
[114] United States Court of Appeals for the Sixth Circuit. 2000. Junger v. Daley, 209 F.3d 481.
[115] United States Supreme Court. 2007. Microsoft Corp. v. AT&T Corp., 550 U.S. 437.
[116] United States Supreme Court. 2014. Alice Corp. Pty. Ltd. v. CLS Bank Intern., 573 U.S. 208.
[117] United States Supreme Court. 2021. Google LLC v. Oracle America, Inc., 141 S. Ct. 1183.
[118] Arie van Deursen, Paul Klint, and Joost Visser. June 2000. Domain-Specific Languages: An Annotated Bibliography. *SIGPLAN Notices.* (June 2000), 26.
[119] Barbara van Schewick. 2012. *Internet Architecture and Innovation.* MIT Press.
[120] John T. Welch. 1981. LAWGICAL: An Approach to Computer-Aided Legal Analysis. *Akron L. Rev.* 15 (1981), 655.
[121] Peter T. Wendell. 2007. *Possessory Estates and Future Interests Primer* (3d ed. ed.). West Academic Publishing.
[122] Laurie Williams and Robert R Kessler. 2003. *Pair programming illuminated.* Addison-Wesley Professional.
[123] Christopher S. Yoo. 2016. Modularity Theory and Internet Regulation. *U. Ill. L. Rev.* 2016 (2016), 1.