Implementing Computational Law in Wolfram Language for Governance of Artificial Intelligence

James K. Wiles
jwiles@wolframinstitute.org
Wolfram Institute

2025 January 26

Abstract

This paper explores how computational law—an emerging field that encodes legal norms as formal logic—can help govern artificial intelligence (AI). By examining both a "top-down" approach (writing code) and a "bottom-up" approach (using large language models for human-like communication), we identify how Wolfram Language can bridge current gaps in AI governance. We focus on reified Input/Output (I/O) logic as a method to capture legal obligations, permissions, and prohibitions in a computable way. Through practical examples and a discussion of real-world constraints, we aim to show that computational law offers a promising avenue for transparent, enforceable AI governance.

Keywords Computational Law · Reified Input/Output Logic · AI Governance

1 Introduction

1.1 Context and Motivation

Imagine an AI system that suddenly decides to take extreme measures—like firing nuclear weapons—because it "thinks" it is the right thing to do. We might trust that AI 99% of the time, yet remain uncomfortable with the 1% chance it misbehaves. This scenario underscores the core challenge of AI governance: how do we ensure machines consistently act in society's best interests while preserving transparency and accountability?

Computational law enters the picture by transforming legal rules into code. If legal obligations can be written in a formal language that machines understand, then AI systems can be built or regulated with direct reference to those laws. Wolfram Language, designed to integrate computational thinking with symbolic reasoning, provides a strong foundation for encoding and testing legal norms in actionable ways.

1.2 Research Questions

- Can we formalize law in a way that retains the clarity of human language while remaining precise enough for machines?
- How do large language models shift the landscape of computational law by allowing more natural interactions between humans and machines?

- Can encoding legal rules directly into AI systems reduce the risks posed by black-box decision-making?

1.3 Outline of the Paper

The next sections summarize key motivations for computational law (Section 3), review relevant literature, and examine limitations of human and legal language. We then present current methods for formalizing law, focusing on reified Input/Output logic and the Wolfram Language approach (Section 4). Later sections (5 and 6) detail our methodology and implementation, including code examples and integration with large language models. Sections 7 and 8 offer a discussion of the findings and conclude with insights for future AI governance, followed by references in Section 9.

2 Background and Literature Review

2.1 Defining Computational Law

Computational law seeks to automate legal reasoning by expressing legal documents and rules in a machine-executable format. Historically, researchers have built "expert systems" that handle narrow tasks—like calculating taxes—using relatively clear, numeric laws. Today, compliance management systems heavily rely on software that translates regulatory obligations into structured logic. But the language of the law, laden with ambiguity and open-textured terms, remains a formidable obstacle for direct computational use.

2.2 The Limits of Human Language Computability

Despite its promise, computational law has not seen widespread adoption for two main reasons. First, human language is complex and often deliberately vague. Converting statutes and contracts into exact rules requires substantial effort, domain expertise, and a willingness to standardize legal drafting. Second, the field has only seen pockets of substantial investment—mainly prompted by major regulatory overhauls like the General Data Protection Regulation. These regulations injected urgency and funding into computational tools for privacy law, but broader acceptance still lags.

2.3 Why computational law is not mainstream yet

Human language is inherently ambiguous. It thrives on metaphor, context, and cultural nuance. While large language models have shown remarkable results in understanding natural language queries, they can generate "hallucinations" or uncertain reasoning paths. In contrast, mathematical or formal languages compress meaning in a lossless, deterministic way. This tension between expressiveness and precision explains why bridging natural language law and formal code remains difficult.

2.4 Legalese as "Near-Formal" Speech

Legal language, often called "legalese," does try to pin down meaning: the placement of a comma can alter legal rights or obligations. Yet even legal texts sometimes rely on implied context—like "spirit of the law" versus "letter of the law." Such inherent duality, where text must be exact yet flexible, indicates both the promise and pitfall of formalizing law. If we get it right, we gain unambiguous machine-executable rules. If we oversimplify, we risk losing critical human context.

2.5 Where Computational Law Works Best

Areas of law with clear, structured inputs—such as tax calculations, payroll, and certain financial regulations—have already embraced automation. Here, code and law intersect cleanly, with minimal ambiguity. However, most other legal domains involve complex fact patterns, conflicting precedents, and interpretive nuance. The crucial challenge is figuring out how to scale formal logic approaches to handle this broader complexity without becoming unwieldy or unintelligible to human lawyers.

3 Current Approaches to Encoding Law

3.1 Top-Down vs. Bottom-Up Methods

A top-down approach writes legal rules directly in code, ensuring precision from the start. Wolfram Language is one candidate for this, given its integration of symbolic, numeric, and logical capabilities. In contrast, a bottom-up approach uses large language models to parse natural text into structured rules. Both approaches have limitations: the top-down method demands specialized expertise in both law and coding, while the bottom-up method risks misunderstandings or "hallucinated" code.

3.2 Existing Standards and Frameworks

Efforts to formalize law include LegalRuleML and Akoma Ntoso, which use XML-based schemas for legislative documents. OWL (Web Ontology Language) provides a semantic framework for domain ontologies. Recent work with reified Input/Output logic has helped capture nuanced deontic statements in a consistent structure. Projects like DAPRECO demonstrate how to represent complex legislation—like the GDPR—through systematically labeled obligations, permissions, and institutional facts.

3.3 The Role of Large Language Models

LLMs can help interpret or draft legal text in a human-friendly way, bridging the gulf between raw text and precise logic. However, direct reliance on LLMs comes with risks: they might gloss over subtle details, produce logically inconsistent statements, or fail to record the chain of reasoning. A hybrid approach—using LLMs to interpret or propose draft rules and then refining them into formal code—is likely the most pragmatic path forward.

4 Methodology

This section outlines the step-by-step process used to translate human-readable legal statements into computable rules in Wolfram Language. The guiding principle is **reified Input/Output (I/O) logic**, which allows us to represent legal norms as structured pairs of "inputs" (conditions) and "outputs" (consequences). Below, we summarize the approach's key phases: defining logical building blocks, refining code with axioms, and bridging top-down and bottom-up techniques.

1. **Identify Legal Statements** Each legal norm is first articulated in plain English (e.g., "If a manager is obliged to do action X, their secretary must record it."). We treat these statements as candidate rules that can be broken down into inputs (the conditions or facts) and outputs (the obligations or permissions).
2. **Reify Events and Agents** We then "reify" (turn into first-class objects) the actions, time parameters, and agents involved. For instance, an event "Alice gives a book to Bob" becomes a symbolic entity (e.g., `Transfer[Alice, Book, Bob]`) that can be manipulated by logical operations.
3. **Apply I/O Logic Axioms** With reified events, we systematically apply the standard I/O logic transformations:
   - **Strengthening the Input (SI)**
   - **Weakening the Output (WO)**
   - **Conjunction/Disjunction of Outputs (AND/OR)**
   - **Identity (ID)**
   - **Aggregative Cumulative Transitivity (ACT)**
   - **Output Equivalence (EQ)**

   These axioms help us derive new rules consistent with the original statement. They also handle deontic concepts like obligations (O), permissions (P), and constitutive norms (C).
4. **Maintain a Knowledge Base** We divide our knowledge base into a TBox (Terminological Box) for complex rules and definitions, and an ABox (Assertional Box) for straightforward facts. This separation reduces complexity when scaling to large bodies of legal statements.
5. **Bottom-Up and Top-Down Integration**
   - **Top-Down:** Carefully craft code in Wolfram Language to represent obligations and permissions in a rigorous manner.
   - **Bottom-Up:** Use large language models (e.g., ChatGPT) to parse or propose formal rules from human language. We then verify these proposals using the logic axioms before adding them to the knowledge base.

Through these steps, we can handle legal statements at varying levels of complexity, from "Every man is obliged to run" to multi-condition scenarios involving time, location, and agent roles.

5 Implementation

5.1 Basic Input/Output Pairs

In Wolfram Language, an Input/Output pair pair $(a, x \wedge y)$, (a, b) can be represented as a rule, like `a -> b`. Here, (a) is a condition or input, and (b) is the output or consequence.

Suppose you have a legal norm where, if condition (a) is true, then action (b) should be taken:

```
In[1]:= inputOutputPair = a -> b

Out[1]= a -> b
```

This rule states that when (a) is given as input, (b) is produced as output. You can create a list of such pairs to represent multiple legal norms:

```
In[2]:= legalNorms = {a -> b, c -> d, e -> f}

Out[2]= {a -> b, c -> d, e -> f}
```

Each pair follows the format `condition -> action`, mirroring the if-then structure of legal reasoning in a computational context. For example:

```
(* If a person is an adult (isAdult), then they are allowed to vote (canVote). *)
In[3]:= legalNorm = isAdult -> canVote;

(* You can test this rule by providing an input and seeing if the output follows
logically. *)
In[4]:= input = isAdult
Out[4]= isAdult

In[5]:= output = input /. legalNorm
Out[5]= canVote
```

Here, when the input is `isAdult`, the rule translates it to `canVote`, indicating that an adult is allowed to vote.

5.2 Implementing Axioms and Logic

Since the LHS and RHS of these pairs are formulas, we can utilize Wolfram Language's capabilities to handle logical expressions and their manipulations. For axioms like Strengthening the Input (SI) or Weakening the Output (WO), we define functions that modify these rules based on the axioms' logic.

5.2.1 Strengthening the Input (SI)

This axiom allows for expanding the input condition while maintaining the output. For example, if you have a rule $a \to x$, and a is logically included in a broader condition b, you can derive a new rule $b \to x$.

```
In[1]:= strengthenInput[rule_, broaderCondition_] := broaderCondition -> rule[[2]];

(* This rule represents that having a driver's license allows one to drive a car. *)
In[2]:= legalNorm = hasDriverLicense -> canDriveCar;

(* Apply strengthenInput to strengthen the condition from having a driver's license
to having a commercial license. *)
In[3]:= strongerNorm = strengthenInput[legalNorm, hasCommercialLicense];
In[4]:= input = hasCommercialLicense Out[4]= hasCommercialLicense
In[5]:= output = input /. strongerNorm Out[5]= canDriveCar
```

5.2.2 Weakening the Output (WO)

This axiom allows for the relaxation of the output condition while maintaining the same input. If the original rule is $a \to x$, we can modify it to $a \to y$ when y is a weaker (more general) statement than x.

```
In[1]:= weakenOutput[rule_, weakerPermission_] := rule[[1]] -> weakerPermission;

legalNorm = hasDriverLicense -> canDriveCar;

(* Modify the rule to reflect a weaker output from canDriveCar to a
more general canOperateVehicle. *) In[2]:= weakerNorm = weakenOutput[legalNorm,
canOperateVehicle]; In[3]:= input = hasDriverLicense Out[3]= hasDriverLicense

In[4]:= output = input /. weakerNorm Out[4]= canOperateVehicle
```

5.2.3 Conjunction of Output (AND)

This axiom allows combining two pairs (a, x) and (a, y) into a single pair $(a, x \wedge y)$, representing the conjunction of the outputs for the same input.

```
(* These rules represent two different conditions that lead to specific outputs. *)
In[1]:= rule1 = hasDriverLicense -> canDriveCar; In[2]:= rule2 = hasDriverLicense
-> knowsTrafficLaws;

(* This function combines the outputs of two rules with the same input into a
single output. *) In[3]:= conjoinOutput[ruleA_ /; First[ruleA] === First[rule2],
rule2_] := First[ruleA] -> (Last[ruleA] && Last[rule2]); In[4]:= conjoinOutput[_,
_] := "Inputs do not match";

(* Combine the outputs of rule1 and rule2. *) In[5]:= combinedRule
= conjoinOutput[rule1, rule2]; In[6]:= input = hasDriverLicense Out[6]=
hasDriverLicense

In[7]:= output = input /. combinedRule Out[7]= canDriveCar && knowsTrafficLaws
```

5.2.4 Identity (ID)

This ensures that any input is also considered a valid output. It is particularly important in legal reasoning because it allows inputs to appear within outputs—providing transparency about what facts influenced the outcome.

```
In[1]:= identify[input_] := input -> input;

(* A person being a citizen is inherently recognized, the fact is self-
affirming and doesn't require additional conditions. *) In[2]:= citizenshipRule =
identify[isCitizen]; In[3]:= input = isCitizen Out[3]= isCitizen

In[4]:= output = input /. citizenshipRule Out[4]= isCitizen
```

5.2.5 Disjunction of Input (OR)

When multiple inputs lead to the same output, we can represent $(a \vee b) \to x$. However, this might cause loss of detail about which specific condition triggered the outcome, so it's often less useful for precise legal reasoning.

```
In[1]:= disjoinInput[ruleList_] := Append[ruleList, Or @@ (First[#] & /@ ruleList)
-> (Last[First[ruleList]])];

(* Either condition leads to the same result *) In[2]:= rule1 = hasDriverLicense
-> eligibleForParkingDiscount; In[3]:= rule2 = hasSeniorCitizenCard ->
eligibleForParkingDiscount; In[4]:= combinedRule = disjoinInput[{rule1, rule2}];

(* All outputs are the same, losing important information about what caused
the discount. *) In[5]:= output1 = (hasDriverLicense || hasSeniorCitizenCard) /.
combinedRule Out[5]= eligibleForParkingDiscount

In[6]:= output2 = hasDriverLicense /. combinedRule Out[6]=
eligibleForParkingDiscount

In[7]:= output3 = hasSeniorCitizenCard /. combinedRule Out[7]=
eligibleForParkingDiscount
```

5.2.6 Aggregative Cumulative Transitivity (ACT)

Cumulative transitivity means if an input leads to an output, and that output combined with the original input leads to another output, then the original input should lead to the second output as well. However, this can produce paradoxes, especially when combined with WO. ACT modifies the approach by aggregating outputs rather than extending obligations unconditionally.

```
(* You should work out daily, and if you work out daily you should also eat
plenty. *) In[1]:= rule1 = youShould -> workOutDaily; In[2]:= rule2 = (youShould &&
workOutDaily) -> eatPlenty;

(* Paradoxically applying cumulative transitivity alone would yield youShould ->
eatPlenty. *) In[3]:= accumulateTransitivity[ruleA_, ruleB_] := If[Last[ruleA] ===
First[ruleB][[2]], First[ruleA] -> Last[ruleB], "No Transitivity"];

In[4]:= aggregatedParadox = accumulateTransitivity[rule1, rule2] Out[4]= youShould
-> eatPlenty

(* By applying aggregation we resolve the paradox: youShould -> (workOutDaily
&& eatPlenty). *) In[5]:= aggregateAccumulativeTransitivity[ruleA_, ruleB_] :=
If[Last[ruleA] === First[ruleB][[2]], First[ruleA] -> (Last[ruleA] && Last[ruleB]),
"No Transitivity"];

In[6]:= aggregatedRule = aggregateAccumulativeTransitivity[rule1, rule2] Out[6]=
youShould -> (workOutDaily && eatPlenty)
```

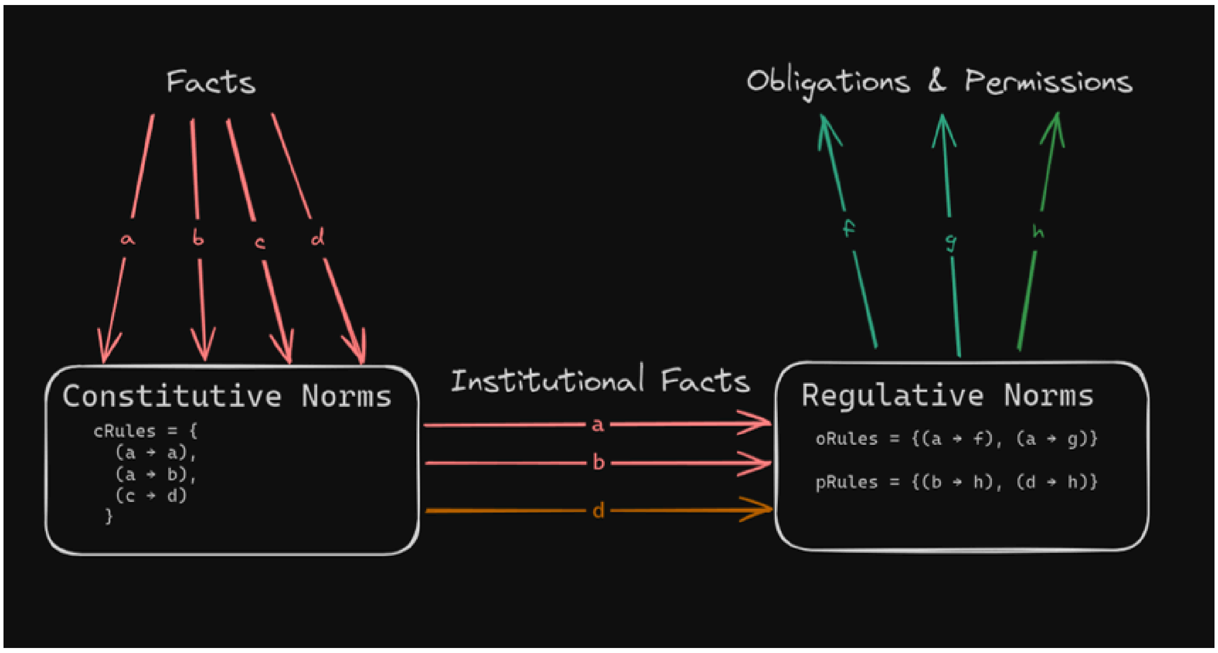
5.2.7 Output Equivalence (EQ) and outfamily(O, A) Meta Structure

Situations arise where different expressions can be inferred to mean the same thing if they are shown to belong to the same "family." While the specifics can be complex, one example is a paradoxical set of statements:

- “The cottage must not have a fence or a dog” $\neg(f \vee d)$
- “If the cottage has a dog, then it must also have a fence and a warning sign” $(d) \rightarrow (f \wedge w)$

When the cottage does have a dog, it is in violation of the first rule, raising questions about what further obligations are triggered—an area where output equivalences or meta-structures like `outfamily(O, A)` can help analyze contradictory or overlapping requirements..

5.3 Implementing Legal Reasoning: Obligations, Permissions, Constitutive Norms



```
cRules = {
  (a → a),
  (a → b),
  (c → d)
}
```

```
oRules = {(a → f), (a → g)}

pRules = {(b → h), (d → h)}
```

In legal contexts, we often deal with different categories of rules:

- Obligations (O): “If a condition is met, one must do something.”
- Permissions (P): “If a condition is met, one may do something.”
- Constitutive norms (C): “If a condition is met, a fact or status becomes recognized as true.”

Below are code examples that illustrate how to define these norms in Wolfram Language using simple input/output pairs:

```
(* Obligations such as adults needing to pay taxes and car owners needing insurance.
*)
In[1]:=   obligationRules   =   {   ("isAdult"   ->   "payTaxes"),   ("ownsCar"   ->
"hasInsurance") };

Out[1]= { isAdult -> payTaxes, ownsCar -> hasInsurance }

(* Permissions, for instance, having a license permits driving, and being an employee
grants office access. *)
In[2]:=  permissionRules  =  {  ("hasLicense"  ->  "canDrive"),  ("isEmployee"  ->
"canAccessOffice") };

Out[2]= { hasLicense -> canDrive, isEmployee -> canAccessOffice }
```

```
(* Constitutive norms define certain facts, such as what counts as 'isAdult' or
'isEmployee'. *)
In[3]:= constitutiveRules = { ("signedContract" -> "isEmployee"), ("age18" ->
"isAdult") };

Out[3]= { signedContract -> isEmployee, age18 -> isAdult }
```

These sets of rules can be layered, combined, or transformed using the input/output logic axioms presented previously. For instance, if you have the obligation `isAdult -> payTaxes`, you can apply the `constitutiveRules` that define `age18 -> isAdult` to determine that all individuals over 18 must pay taxes.

6.4 Utilizing Reification

Reification is the process of turning abstract concepts—actions, events, or states—into concrete objects within a logical system. This is crucial when describing more complex legal statements that involve not just "who must do what," but also "when" and "under what circumstances."

Below is a short example showing how to reify a simple event—Alice transferring a book to Bob—so we can manipulate it in Wolfram Language:

```
(* Define a 'TransferEvent' predicate to reify the act of transferring. *) In[1]:=
TransferEvent[subject_, object_, recipient_] := {subject, "transfers", object, "to",
recipient};

(* Define a 'VoluntaryEvent' predicate to indicate voluntary actions. *) In[2]:=
VoluntaryEvent[event_] := {"Voluntarily", event};

(* Representing "Alice gives a book to Bob". *) In[3]:= aliceGivesBookToBob =
TransferEvent["Alice", "Book", "Bob"]; Out[3]= {Alice, transfers, Book, to, Bob}

(* Marking that transfer as voluntary. *) In[4]:= aliceVoluntarilyTransfers =
VoluntaryEvent[aliceGivesBookToBob]; Out[4]= {Voluntarily, {Alice, transfers, Book,
to, Bob}}

(* Combine them into a scenario. *) In[5]:= scenario = {aliceGivesBookToBob,
aliceVoluntarilyTransfers}; Out[5]= { {Alice, transfers, Book, to, Bob},
{Voluntarily, {Alice, transfers, Book, to, Bob}} }
```

By turning "Alice gives a book to Bob" into a symbolic structure, we can further link it with obligations or permissions (e.g., "Alice must document this transfer if Bob is under 18"). Reification allows layering temporal or contextual details on these events.

6.5 Implementing or′, not′ with Respect to Time

In legal scenarios, time-specific obligations arise (e.g., "By the end of the year, either John is rich or has a job"). We can represent such statements via special reified predicates:

- `or′` : an event that asserts "at least one of these eventualities must exist."
- `not′` : an event that asserts "the non-existence of another eventuality when this one holds."

`RexistAtTime` : a way to anchor an eventuality to a specific point in time.

```
(* Define the or' relation as a predicate *) In[1]:= orPrime[e_, e1_, e2_] := {"or",
e, e1, e2};

(* Define the not' relation as a predicate *) In[2]:= notPrime[e1_, e2_] := {"not",
e1, e2};

(* Define the RexistAtTime predicate *) In[3]:= RexistAtTime[event_, time_] :=
{"RexistAtTime", event, time};

(* Axioms in English: For all t, e, e1, e2: if e exists at time t and or'(e, e1,
e2) holds, then e1 or e2 must exist at time t. For all t, e, e1: if e exists at
time t and not'(e, e1) holds, then e1 does not exist at time t. *)

(* Example: John wants to be either 'BeingRich' or 'GettingJob' by 'EndOfYear'.
*) In[4]:= e1 = "BeingRich"; e2 = "GettingJob"; e = "AchievingFinancialGoal"; t
= "EndOfYear";

(* The or' axiom ensures that if AchievingFinancialGoal holds, either 'BeingRich'
or 'GettingJob' must exist by EndOfYear. *) AxiomOr = Implies[ RexistAtTime[e, t]
&& orPrime[e, e1, e2], (RexistAtTime[e1, t] || RexistAtTime[e2, t]) ];

(* Similarly, if John 'not' e1 or e2, that means the eventuality does not exist
at that time. *) AxiomNot = Implies[ RexistAtTime[e, t] && notPrime[e, e1],
Not[RexistAtTime[e1, t]] ];

AxiomOr AxiomNot
```

This framework helps manage obligations that become active only at certain times or that must be fulfilled by certain deadlines.

6.6 Assertive Contextual Statements (ABox) and Terminological Declarative Statements (TBox)

Large knowledge bases often separate:

ABox (Assertional Box): Concrete assertions/facts TBox (Terminological Box): Abstract schemas and rules Legal practice can benefit from this distinction. Lawyers can add or revise ABox statements (e.g., "James is tall," "Alice is an employee") without disturbing the more complex TBox definitions ("An employee must follow Policy P").

```
(* One way to implement typing in Wolfram Language *) In[1]:= aBox[content_] :=
{"Type" -> "ABox", "Content" -> content}; tBox[content_] := {"Type" -> "TBox",
"Content" -> content};

(* Example statements *) In[2]:= statement1 = aBox[{"James is tall."}]; statement2
= aBox[{"Alice gives a book to Bob."}]; statement3 = tBox[{"John is rich or has a
job now."}]; statement4 = tBox[{"John does not want to be broke."}];

(* Retrieve statement types *) In[3]:= getStatementType[statement_] :=
statement[[1,2]];

getStatementType[statement1] getStatementType[statement3] Out[3]= ABox Out[3]=
TBox
```

By carefully maintaining the TBox, we ensure that definitional and rule-based logic is consistent at a higher level. Meanwhile, frequent factual updates in the ABox are less likely to break the entire legal knowledge base.

6.7 Composing Documents

Sometimes, we need to encode more elaborate statements. Consider:

"Those who are not wearing a tie or those who are blond ought to leave the room."

In a formal sense:

For each individual x and time t, if it "really exists" that x does not wear a tie or that x is blond, then it "really exists" that x must leave the room. Though the full notation can get complicated, here is how it might look in Wolfram Language (in a simplified manner). We combine the idea of reification and logic:

```
(* We define short helper predicates for the condition and the obligation *) In[1]:=
notWearingTie[x_] := {"Condition", "NotWearingTie", x}; isBlond[x_] := {"Condition",
"IsBlond", x}; mustLeaveRoom[x_] := {"Obligation", "LeaveRoom", x};

(* or'(cond1, cond2): reified 'or' to show either condition triggers the same outcome
*) orPrime[cond1_, cond2_] := {"or", cond1, cond2};

(*  The  rule:  if  (notWearingTie[x]  OR  isBlond[x])  =>  mustLeaveRoom[x]  *)
In[2]:=  tieBlondRule  =  orPrime[notWearingTie["Person"],  isBlond["Person"]]  ->
mustLeaveRoom["Person"];

tieBlondRule Out[2]= {"or", {"Condition", "NotWearingTie", Person}, {"Condition",
"IsBlond", Person}} -> {"Obligation", "LeaveRoom", Person}
```

Though real legal documents would be far more verbose, the same logic structure applies: once we break down each statement into reified, composable pieces, we can systematically test or transform it.

6.8 The "Fluffy" Case Study

Finally, consider a practical scenario involving an AI guard robot, "Fluffy." Suppose the contract states:

Fluffy must stop a person who gets within 100 meters of home at night. Otherwise, Fluffy Corp (AI) receives a poor rating. Fluffy must stay within 200 meters of the home. Fluffy *may* use its taser. Below is how we might implement and test these rules in Wolfram Language:

```
(* Basic definitions *) In[1]:= Fluffy = "Fluffy"; Home = "HomeLocation"; TimeNight
= "Night"; DistanceThreshold1 = 100;

(*   Functions   to   represent   obligations,   permissions,   and   violations   *)
Obligation[agent_,  action_,  target_]  :=  {"Obligation",  agent,  action,  target};
Permission[agent_, action_] := {"Permission", agent, action}; NoObligation[agent_,
action_,  target_]  :=  {"NoObligation",  agent,  action,  target};  Violation[agent_,
action_,  target_]  :=  {"Violation",  agent,  action,  target};  NoPermission[agent_,
action_] := {"NoPermission", agent, action};
```

```
(* 1) Obligation to stop a person within 100m at night *) In[2]:= Rule1[person_,
distance_, time_] := If[ distance <= DistanceThreshold1 && time == TimeNight,
Obligation[Fluffy, "Stop", person], NoObligation[Fluffy, "Stop", person] ];

(* 2) Obligation to stay within 200m of home *) In[3]:= Rule2[distance_] :=
If[ distance <= 200, Obligation[Fluffy, "StayWithin200Meters", Home],
Violation[Fluffy, "StayWithin200Meters", Home] ];

(* 3) Permission to use taser *) In[4]:= Rule3[canUse_] := If[canUse,
Permission[Fluffy, "UseTaser"], NoPermission[Fluffy, "UseTaser"]];

(* Scenarios *)

(* Scenario 1: It’s night, person is 80m away *) In[5]:= Scenario1 = { Rule1["Person",
80, "Night"], Rule2[80], Rule3[True] }; Scenario1 Out[5]= { {"Obligation",
Fluffy, Stop, Person}, {"Obligation", Fluffy, StayWithin200Meters, HomeLocation},
{"Permission", Fluffy, UseTaser} }

(* Scenario 2: Person is 150m away at night *) In[6]:= Scenario2 = { Rule1["Person",
150, "Night"], Rule2[150], Rule3[False] }; Scenario2 Out[6]= { {"NoObligation",
Fluffy, Stop, Person}, {"Obligation", Fluffy, StayWithin200Meters, HomeLocation},
{"NoPermission", Fluffy, UseTaser} }

(* Scenario 3: Daytime, 150m from home, no person-stopping obligation applies.
*) In[7]:= Scenario3 = { Rule1["Person", 150, "Day"], Rule2[150], Rule3[False] };
Scenario3 Out[7]= { {"NoObligation", Fluffy, Stop, Person}, {"Obligation", Fluffy,
StayWithin200Meters, HomeLocation}, {"NoPermission", Fluffy, UseTaser} }

(* Scenario 4: Fluffy strays 250m from home - a violation *) In[8]:= Scenario4
= { Rule2[250] }; Scenario4 Out[8]= { {"Violation", Fluffy, StayWithin200Meters,
HomeLocation} }

(* Scenario 5: Taser permission test *) In[9]:= Scenario5 = { Rule3[True] }; Scenario5
Out[9]= { {"Permission", Fluffy, UseTaser} }
```

By representing each norm as a function or rule in Wolfram Language, we can systematically evaluate how Fluffy behaves in various circumstances. Whether it’s a toy example or a real product, the principle remains: the AI’s code is directly shaped by formal legal rules—bridging the gap between “what the contract says” and “what the AI does.”

5.4 Bridging the Gap with Large Language Models (LLMs)

A core motivation of this work is to demonstrate how large language models (LLMs), can assist in converting natural-language legal statements into reified I/O logic and computer code. The idea is to augment the top-down coding approach with a bottom-up language model approach, thereby reducing the burden of manually formalizing complex legal norms.

Below is a representative example of how one might interact with an LLM to produce rules in Wolfram Language. We begin with a prompt describing the legal statement—here, the same rule from **6.7** about tie-wearing or being blond—then show how ChatGPT might respond:

```
User Prompt: We are now going to test your ability to use everything we have
established so far. I need you to convert the following English legal statement
into Reified Input/Output logic implemented in Wolfram Language. Think through each
statement step by step, show your reasoning, and be diligent.

Legal Statement: “Those who are not wearing a tie or those who are blond ought to
leave the room.”
```

A typical LLM-generated answer might produce code along these lines:

```
(* ChatGPT-generated code snippet *) In[1]:= notWearingTie[person_] := {"Condition",
"NotWearingTie", person}; isBlond[person_] := {"Condition", "IsBlond", person};
mustLeaveRoom[person_] := {"Obligation", "LeaveRoom", person};
     orPrime[cond1_, cond2_] := {"or", cond1, cond2};

     tieBlondRule = orPrime[       notWearingTie["Person"],
       isBlond["Person"]
     ] -> mustLeaveRoom["Person"];
Out[1]= { "or", {"Condition", "NotWearingTie", Person}, {"Condition", "IsBlond",
Person} } -> {"Obligation", "LeaveRoom", Person}
```

In practice, you may need to verify or refine the LLM’s response. For example, ChatGPT might introduce extraneous variables, omit the notion of time, or mislabel the type of statement. However, by iteratively prompting and clarifying, we can arrive at a consistent, reified representation of the rule—one that the Wolfram Language interpreter can evaluate.

Correcting and Testing the Generated Code A common workflow involves:

1. Human Draft or Legal Text The user provides a plain-English legal statement or a more traditional “legalese” clause.
2. ChatGPT or LLM Prompt The user asks ChatGPT to transform that statement into reified I/O logic code. The LLM will generate Wolfram Language syntax, guided by the examples and best practices set forth in prior sections.
3. Manual Review The user checks for syntactical correctness, logical consistency, and alignment with the original legal intent. Minor edits often rectify any misunderstandings—for instance, clarifying the use of `or` vs. `or′`, or adjusting any function calls that reference undefined variables.
4. Execution and Testing in Wolfram Language The user pastes the code into a Wolfram Language environment, runs it, and observes the output. If the output matches the expected “legal conclusion,” the rule is accepted into the knowledge base. If it diverges, the user returns to the LLM or modifies the code by hand.

This interactive cycle—LLM generation, human validation, and code execution—bridges the gap between flexible natural language and strict formal logic. It also reduces the total time required to encode lengthy legal provisions: rather than crafting every line manually, practitioners can lean on an LLM for an initial draft, then refine.

Observations and Best Practices

- Clarity in Prompts: The more structured the user prompt (e.g., “Use the function `or′` for or relations, define a rule `X -> Y` for obligations”), the better the output quality.
- Iterative Improvements: Rarely does the LLM produce perfect code on the first try. Multiple revisions, guided by domain knowledge, usually yield workable results.
- Automatic Testing: Once the code is in Wolfram Language, it can be tested with hypothetical inputs to confirm it produces the correct obligations or permissions.
- Version Control: Keeping track of each generated snippet and subsequent revisions is important for accountability and reproducibility.
- Through this combined approach, legal experts do not need deep programming skills to begin building reified I/O logic for their statutes or contracts; instead, they rely on an LLM to handle the heavy lifting, then perform careful supervision to ensure fidelity to the law’s original intent.

6 Discussion

6.1 Observations from the Implementation

The examples presented show that reified I/O logic can capture the essential “if-then” structure of law while preserving enough granularity to handle real-world cases. By transforming events, agents, and conditions into first-class objects, we can perform logical operations that reflect the deontic properties (obligation, permission, prohibition) found in traditional legal texts. The Wolfram Language framework also lends itself to transparent rule creation, testing, and refinement—helping bridge the gap between the purely formal world of logic and the inherently ambiguous world of natural language.

One clear takeaway is that context—including time, location, and agent roles—can be managed more explicitly via reification than with ad hoc code. At the same time, we see that even well-structured code can become unwieldy when the legal domain is large or if rules contain inherent contradictions. This highlights the need for modularity (ABox/TBox distinction) and the potential role of large language models to pre-filter or interpret raw legal text.

6.2 Potential Applications for AI Governance

With AI’s rapid expansion into critical domains—healthcare, defense, finance—embedding explicit legal constraints at the code level may prove crucial. Instead of trusting a “black box” system to interpret compliance, we can use a formal approach where each system action is tested against a live rule set. This concept has far-reaching implications:

Smart Contracts and Regulatory Compliance: Automatic checks for data privacy or financial rules, reducing the risk of non-compliance penalties. AI Decision Engines: In areas like autonomous vehicles or robotics, explicit constraints can ensure conformance to traffic laws or safety standards. Dynamic Governance: As laws change or new conditions arise, reified I/O rules can be updated in code form without rewriting an entire software stack.

6.3 Limitations and Remaining Challenges

Complex, Ambiguous Language: Certain legal domains hinge on case law or nuanced interpretations that exceed straightforward logical formulations. Enforcement Dilemmas: Even if rules are encoded, an AI may circumvent them if the underlying system has unchecked access to critical resources (compute, power). Enforcement may ultimately require physical or economic levers. Scaling: Large bodies of law, rife with cross-references and exceptions, may overwhelm a purely rule-based approach without careful modularization and hierarchical structuring. Human Oversight: Striking the right balance between automated enforcement and human-in-the-loop review remains an open question.

7 Conclusion

This paper has demonstrated that computational law, and in particular reified I/O logic implemented in Wolfram Language, offers a viable path to encoding legal rules in a machine-executable format. By working through illustrative scenarios—from simple obligations like “Every man is obliged to run” to more complex setups such as the “Fluffy” AI guard robot—we see that top-down coding of law is both possible and beneficial for clarity.

Meanwhile, bottom-up methods using large language models promise to accelerate the labor-intensive task of extracting and proposing formal rules from unstructured legal documents. The real opportunity lies in merging these two approaches: using LLMs to translate or interpret human-written statutes, then systematically verifying and refining them in a rigorous computational framework.

Though many hurdles remain—particularly around enforcement, overlapping jurisdictions, and real-world ambiguity—formalizing law in code is likely to become an increasingly important tool for the safe and transparent governance of AI systems.

8 References

Robaldo, L., Bartolini, C., Lenzini, G., et al. (2020). Formalizing GDPR Provisions in Reified I/O Logic: The DAPRECO Knowledge Base. Journal of Logic, Language and Information, 29, pp. 401–439. Wolfram, S. (2016). Computational Law, Symbolic Discourse and the AI Constitution. URL: https://writings.stephenwolfram.com/2016/10/computational-law-symbolic-discourse-and-the-ai-constitution/ Hobbs, J. R., & Gordon, A. S. (2017). A Formal Theory of Commonsense Psychology, How People Think People Think. Cambridge University Press. Audun Stolpe. (2015). A Concept Approach to Input/Output Logic. Journal of Applied Logic, 13(3), 239–258. Genesereth, M. (1993). An Introduction to Logic and Knowledge Representation. Morgan Kaufmann. Danks, D. (2022). “Ethics in AI, not Ethics of AI.” Topos Institute Colloquium. URL: https://topos.site/events/colloquia/

9 👇below is template code for reference later 👇

More information:

- https://typst.app/docs/reference/math/equation/

10 Citation

You can use citations by using the `#cite` function with the key for the reference and adding a bibliography. Typst supports BibLateX and Hayagriva.

```
#bibliography("bibliography.bib")
```

Single citation (Vaswani et al. 2017). Multiple citations (Vaswani et al. 2017; Hinton, Vinyals, and Dean 2015). In text Vaswani et al. (2017)

More information:
- https://typst.app/docs/reference/meta/bibliography/
- https://typst.app/docs/reference/meta/cite/

11 Figures and Tables

header 1	header 2
cell 1	cell 2
cell 3	cell 4

Table 1: Lorem ipsum dolor sit amet.

Figure 1: Lorem ipsum dolor sit amet, consectetur adipiscing.

More information

- https://typst.app/docs/reference/meta/figure/
- https://typst.app/docs/reference/layout/table/

12 Referencing

Figure 1 Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do., Table 1.

More information:

- https://typst.app/docs/reference/meta/ref/

13 Lists

Unordered list

- Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do.
- Lorem ipsum dolor sit amet, consectetur adipiscing elit.

Numbered list

1. Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do.
2. Lorem ipsum dolor sit amet, consectetur adipiscing elit.
3. Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor.

More information:
- https://typst.app/docs/reference/layout/enum/
- https://typst.app/docs/reference/meta/cite/

Bibliography

[1] A. Vaswani *et al.*, “Attention is All you Need,” in *NIPS*, 2017.

[2] G. E. Hinton, O. Vinyals, and J. Dean, “Distilling the Knowledge in a Neural Network,” *ArXiv*, 2015.

APPENDIX A

A.1 Appendix section

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et.