An End-to-End Pipeline from Law Text to Logical Formulas

Downloaded from: https://research.chalmers.se, 2026-11-11 09:45 UTC

Citation for the original published paper (version of record):
Ranta, A., Listenmaa, I., Soh, J. et al (2022). An End-to-End Pipeline from Law Text to Logical Formulas. Frontiers in Artificial Intelligence and Applications, 362: 237-242.
http://dx.doi.org/10.3233/FAIA220473

N.B. When citing this work, cite the original published paper.

research.chalmers.se offers the possibility of retrieving research publications produced at Chalmers University of Technology. It covers all kind of research output: articles, dissertations, conference papers, reports etc. since 2004. research.chalmers.se is administrated and maintained by Chalmers Library

(article starts on next page)

Legal Knowledge and Information Systems
E. Francesconi et al. (Eds.)

© 2022 The authors and IOS Press.
This article is published online with Open Access by IOS Press and distributed under the terms of the Creative Commons Attribution Non-Commercial License 4.0 (CC BY-NC 4.0).
doi:10.3233/FAIA220473

An End-to-End Pipeline from Law Text to Logical Formulas

Aarne RANTA [a,1], Inari LISTENMAA [b], Jerrold SOH [b], Meng Weng WONG [b]
[a] *Chalmers University of Technology and University of Gothenburg*
[b] *Centre for Computational Law, Singapore Management University*

Abstract. We propose a pipeline for converting natural English law texts into logical formulas via a series of structural representations. Text texts are first parsed using a formal grammar derived from light-weight annotations. An intermediate representation called assembly logic is then used for logical interpretation and supports translations to different back-end logics and visualisations. The approach, while rule-based and explainable, is also robust: it can deliver useful results from day one, but allows subsequent refinements and variations.

Keywords. legal formalisms, legal text parsing, Grammatical Framework

1. Introduction

Expressing laws computably is a classic objective of AI & Law [1] and a prerequisite to automating downstream tasks such as compliance checking [2], policy support [3], legislative simulation [4], and formal verification [3]. But faithfully translating law to logic is challenging [5], often requiring expertise in both legal and formal methods. This "natural language barrier" [6] poses a significant "knowledge bottleneck" [7] to computational law. Numerous strategies have been devised for bridging the gap. These include domain-specific ontologies [8], intermediate formalisms [6], and specialised human workflows [8,9]. Early on, [10] had already imagined automatic parsers for translating laws into logic. Several steps have been taken towards that vision. McCarty [6] used [11]'s statistical parser to extract from judicial opinions syntax trees then converted into semantic representations. [12] extract formal rules from deontically- and structurally-annotated legal texts with the standard NLP parsers, while [5] experiment with neural semantic parsing and open relation extraction.

However, whether the chosen framework accommodates the logic representation desired is not always clear [13]. This paper contributes a partially-automated

[1]Corresponding Author: Aarne Rante, aarne.ranta@cse.gu.se. This research is supported by the National Research Foundation (NRF), Singapore, under its Industry Alignment Fund — Pre-Positioning Programme, as the Research Programme in Computational Law. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

law to logic pipeline based on Grammatical Framework (GF, [14]). While prior legal GF applications [15,16] focused on Controlled Natural Languages (CNL, e.g., [17,18]), our application tackles real-world law texts, albeit still exploiting key GF features such as modularity, precision, and support for semantic back-ends via an abstract syntax. We develop a method for automatically extracting a grammar from light-weight annotations which non-experts can create. This grammar is usable as-is for a rough analysis of law texts but can also be manually improved. Our code is available open-source.[2]

2. Methodology

The initial input is a statutory text in natural language which we assume has been tokenised by some standard tool. The tokenised text is converted to abstract syntax trees (ASTs) line by line using the GF parser driven by a grammar (Section 2.1). The ASTs are converted into an intermediate representation called assembly logic (Section 2.2) using the Haskell-based methodology from [19]. Assembly logic is more abstract than ASTs from the parser, but preserves more distinctions than standard back-end logics. These distinctions are useful for deriving different output formats such as downstream logics and visualisations (Section 2.3).

2.1. From Law Text to ASTs

GF parsers are driven by grammars, such as GF's general-purpose Resource Grammar Library (RGL, [20]). However, the RGL is insufficient for law texts, which contain special constructs that are essential for the logical structure, such as itemised lists and indented paragraphs. Thus we developed a tailored grammar on top of the RGL. Figure 1 illustrates the grammar building workflow, which adopts a data-driven, top-down approach starting from the text itself. We developed a semi-automated method for grammar writing based on user annotations. The annotations are based on the RGL's grammatical categories (e.g. `NP`, `VP`, `VP2`, `CN`) which are well-known in NLP. Annotators may further specify their own categories, such as `Line`, `Item` and `Ref`, and any categories added will be added to the custom grammar. Completely novel categories *could* be created, but if they deviate too much from the RGL, the resulting grammar cannot leverage the RGL as much. Finally, a Haskell script generates GF rules from the annotated text. The rules are generated in a context-free format that GF can process.

2.2. From ASTs to Assembly Logic

Assembly logic is an intermediate representation between ASTs and standard logics. It is designed to preserve enough syntactic structure to generate representations that humans can easily relate to the original text. For example, it distinguishes between ordinary and reverse implications ("if A then B" vs. "B if A") and preserves quantified noun phrases as units (e.g. "'any organisation"). It is

[2] https://github.com/smucclaw/sandbox/tree/default/aarne#readme

A line in the raw text:

(2) without limiting subsection (1)(a), a data breach is deemed to result in significant harm to an individual —

The line annotated with marks for terminals (#) and nonterminals (*):

```
*Item (2) #without #limiting #subsection *Ref (1)(a) #,
#a *CN data breach #is #deemed #to
*VP result in significant harm to an individual #-
```

Grammar rules derived automatically by the script:

```
Line ::= Item "without" "limiting" "subsection" Ref ","
         "a" CN "is" "deemed" "to" VP "-" ;
Item ::= "(2)" ;
Ref  ::= "(1)(a)" ;
CN   ::= "data" "breach" ;
VP   ::= "result" "in" "significant"
         "harm" "to" "an" "individual" ;
```

The VP rule above refined into more general rules:

```
VP2 ::= "result" "in" NP ;
NP  ::= "significant" "harm" "to" NP ;
NP  ::= "an" CN ;
CN  ::= "individual" ;
```

Figure 1. The grammar extraction process.

intended to be sufficient for this purpose, so that when the grammar is extended (e.g., new law texts), the assembly logic and its back-ends can be kept constant.

Figure 2 shows a sample of the assembly logic implemented as a Haskell datatype `Formula`. It also shows part of an **interpretation function** [19], `iNP`, which converts ASTs of GF type `NP` (Noun Phrase) to assembly logic. These functions use pattern matching over trees. Each AST constructor may have its own pattern, such as for `NP_any_CN` in Figure 2. When the grammar is extended, new patterns can be added. But even if this is not done, the function can take care of the new constructors by the catch-all case (`_`) which treats the new expressions as atomic. Atomic expressions can then be converted to atomic formulas or constants in logics and to single cells in spreadsheets (see Section 2.3).

2.3. From Assembly Logic to Downstream Logics or Visualisations

Assembly logic is mapped into many-sorted logic and then into ordinary predicate logic in TPTP notation [21]. We use many-sorted logic as it better supports compositional translation. Quantification expressed by noun phrases (e.g. "any organisation") are compositionally interpreted as quantifiers with sorts rather than divided into unsorted quantifiers and sort predicates, whereas definite noun phrases (e.g. "that organisation") are interpreted as Russell's iota terms (we write $\iota(A)$ instead of $(\iota x)A(x)$, leaving possible variable bindings to A itself, as is customary in higher-order logic). Both sorted quantifiers and iota terms are eliminated in the conversion from many-sorted to ordinary predicate logic. Iota terms are eliminated in a pass that looks for non-iota terms in their context of use.

Some assembly logic constructors:

```
data Cat =
  CProp | CSet | CInd | ...
data Formula =
  Atomic Cat Atom
| Implication Formula Formula
| Conditional Formula Formula     -- reverse implication
| Quantification String Formula   -- quantifier + domain
```

Semantics of ASTs in the assembly logic:

```
iNP :: Env -> NP -> Formula
iNP env np = case np of
  NP_any_CN cn -> Quantification "ANY" (iCN env cn)
  NP_each_CN cn -> Quantification "EACH" (iCN env cn)
  ...
  _ -> Atomic CInd (toAtom env np) -- convert to string
```

Figure 2. Data structures and conversions related to the assembly logic.

Below is a minimal example, with an existential quantifier in the antecedent and definite noun phrase referring to it in the succedent: "if a notification is a data breach, the notification is affected". Its compositional interpretation in many-sorted logic with iota terms is $(\exists x : \mathit{notification})\mathit{data_breach}(x) \supset \mathit{affected}(\iota(\mathit{notification}))$. When converted to ordinary predicate logic, the existential quantifier is changed into a universal one with a wide scope of implication, and the iota term is interpreted as the bound variable:

```
![X]:(notification(X) => data_breach(X) => affected(X))
```

The AST can also be automatically visualised in a spreadsheet (see Figure 3) displaying the formula trees in a structured format. The spreadsheet format, which serves as the input to a low-code programming platform, is currently under development and will be more fully described in future work.

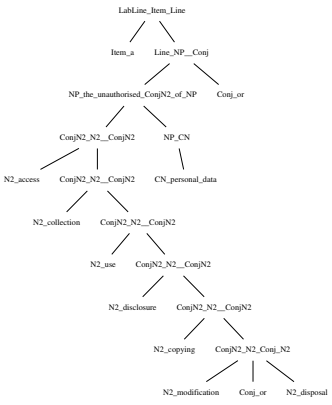
3. Formalizing the Personal Data Protection Act

We illustrate our pipeline using Part 6A of the PDPA, which comprises 47 lines, 1053 tokens, 228 unique tokens. Figure 3 below illustrates the pipeline as applied to one paragraph. The PDPA is Singapore's primary data protection statute and Part 6A governs data breach notifications. While the PDPA has not been examined in AI & Law literature, its subject matter connects it to prior work on the General Data Protection Regulation [8,2]. Part 6A is also complex enough to demonstrate the utility of a computational law approach. Modelling these rules surfaced a race condition in the PDPA: an organisation which promptly notifies both the regulator and the affected individuals of a data breach, as s 26D PDPA generally requires, might violate s 26D(6) which provides that organisations should *not* inform affected individuals if the regulator so directs. A more complete formalism of Part 6A can be found on our code repository.

A paragraph in the raw text:

> "data breach", in relation to personal data, means —
>
> (*a*) the unauthorised access, collection, use, disclosure, copying, modification or disposal of personal data; or
>
> (*b*) the loss of any storage medium or device on which personal data is stored in circumstances where the unauthorised access, collection, use, disclosure, copying, modification or disposal of the personal data is likely to occur.

AST of line (a) and spreadsheet visualization of the paragraph



Logical formula in TPTP notation:

```
![X]:(data_breach(X) & ?[Y]:(personal_data(Y) & IN_RELATION_TO(X,Y)) <=>
(personal_data(X) & ?[Y]:((access(Y,X) | collection(Y,X) | use(Y,X) | disclosure(Y,X) |
copying(Y,X) | modification(Y,X) | disposal(Y,X)) & unauthorized(Y))) | (((storage_medium(X) |
device(X)) & (personal_data(X) & ?[Y]:((circumstances(Y) & (((unauthorized(Y) & (access(Y) |
collection(Y) | use(Y) | disclosure(Y) | copying(Y) | modification(Y) | disposal(Y))) &
is_likely_to_occur(Y))) & is_stored_in(X,Y)))) & loss(X))))
```

Figure 3. An example through the pipeline

4. Conclusion

This paper proposed a pipeline which parses legal text into ASTs using the GF grammar formalism, an intermediate assembly logic, and finally predicate logic. Some pipeline steps can work out of the box when the input scope is extended. The main things to be added are text annotations for extending the grammar, and the conversion of the new grammar rules to the assembly logic. These steps are light-weight enough to make the system feasible to apply to new texts. Further, since GF's mapping between ASTs and natural language is fully reversible, the pipeline can be extended to support natural language generation. Once parsed into GF trees, the source text can be converted into novel forms: declarative sentences can become questions, negations, hypotheticals, etc. That said, this work is a proof of concept and has a some limitations. Importantly, we have not evaluated the accuracy of our PDPA formalisation and aim to do so in future work. A proper evaluation would implicate gold standards developed by human legal and technical experts and vetted by the relevant regulatory body. The legal

language barrier is far from solved, but we hope to have taken one more step towards realising that vision.

References

[1] Sergot MJ, Sadri F, Kowalski RA, Kriwaczek F, Hammond P, Cory HT. The British Nationality Act as a logic program. Communications of the ACM. 1986 May;29(5):370-86.

[2] Hickey D, Brennan R. A GDPR International Transfer Compliance Framework Based on an Extended Data Privacy Vocabulary (DPV). In: Proceedings of JURIX. IOS Press; 2021. p. 161-70.

[3] Haan ND. TRACS: A Support Tool for Drafting and Testing Law. In: Proceedings of JURIX; 1992. p. 63-70.

[4] Bench-Capon TJM. Support for Policy Makers: Prospects for Knowledge Based Systems. In: Proceedings of JURIX; 1992. p. 41-50.

[5] Ferraro G, Lam HP, Tosatto SC, Olivieri F, Islam MB, Beest Nv, et al. Automatic extraction of legal norms: Evaluation of natural language processing tools. In: JSAI International Symposium on Artificial Intelligence. Springer; 2019. p. 64-81.

[6] McCarty LT. Deep semantic interpretations of legal texts. In: Proceedings of ICAIL; 2007. p. 217-24.

[7] Nazarenko A, Lévy F, Wyner A. A Pragmatic Approach to Semantic Annotation for Search of Legal Texts – An Experiment on GDPR. In: Proceedings of JURIX. IOS Press; 2021. p. 23-32.

[8] Palmirani M, Martoni M, Rossi A, Robaldo L. Legal Ontology for Modelling GDPR Concepts and Norms. In: Proceedings of JURIX; 2018. p. 91-100.

[9] Witt A, Huggins A, Governatori G, Buckley J. Converting copyright legislation into machine-executable code: interpretation, coding validation and legal alignment. In: Proceedings of ICAIL. São Paulo Brazil: ACM; 2021. p. 139-48.

[10] Bing J. Designing text retrieval systems for conceptual searching. In: Proceedings of ICAIL. Boston, Massachusetts, United States: ACM Press; 1987. p. 43-51.

[11] Collins M. Head-Driven Statistical Models for Natural Language Parsing. Computational Linguistics. 2003 Dec;29(4):589-637.

[12] Dragoni M, Villata S, Rizzi W, Governatori G. Combining natural language processing approaches for rule extraction from legal documents. In: AI Approaches to the Complexity of Legal Systems. Springer; 2015. p. 287-300.

[13] Wyner A, Governatori G. A Study on Translating Regulatory Rules from Natural Language to Defeasible Logic. In: Proceedings of the 7th International Web Rule Symposium; 2013. p. 16.1-16.8.

[14] Ranta A. Grammatical Framework: Programming with Multilingual Grammars. Stanford: CSLI Publications; 2011.

[15] Angelov K, Camilleri J, Schneider G. A Framework for Conflict Analysis of Normative Texts Written in Controlled Natural Language. The Journal of Logic and Algebraic Programming. 2013;82:216-40.

[16] Digital Grammars, Signatu. GDPR Lexicon; 2018. https://gdprlexicon.com/.

[17] Fuchs NE, Kaljurand K, Kuhn T. Attempto Controlled English for Knowledge Representation. In: Reasoning Web, Fourth International Summer School 2008. 5224. Springer; 2008. p. 104-24.

[18] Ranta A, Angelov K. Implementing Controlled Languages in GF. In: Proceedings of CNL-2009, Marettimo. vol. 5972 of LNCS; 2010. p. 82-101.

[19] Ranta A. Translating between Language and Logic: What Is Easy and What Is Difficult. In: Automated Deduction – CADE-23. Springer Berlin Heidelberg; 2011. p. 5-25.

[20] Ranta A. The GF Resource Grammar Library. Linguistics in Language Technology. 2009;2.

[21] Sutcliffe G. The TPTP problem library and associated infrastructure. Journal of Automated Reasoning. 2009;43(4):337-62.