Extensions to Justification Theory

Simon Marynissen[1,2][0000−0002−2894−1377]

[1] KU Leuven, Leuven, Belgium
[2] Vrije Universiteit Brussel, Brussels, Belgium

1 Background

Justification theory [6] is a unifying framework for semantics of non-monotonic logics. It is built on the notion of a justification, which intuitively is a graph that explains the truth value of certain facts in a structure. Knowledge representation languages covered by justification theory include logic programs, argumentation frameworks, inductive definitions, and nested inductive and coinductive definitions. In addition, justifications are also used for implementation purposes. They are used to compute unfounded sets in modern ASP solvers [10,4], can be used to check for relevance of atoms in complete search algorithms [11], and recent lazy grounding algorithms are built on top of them [5,2].

Semantics in justification theory are determined by *branch evaluations*. These are functions that map chains of facts to a single fact, in most case true, false or unknown. For instance, in well-founded semantics, a positive loop is not allowed in a model, and thus the branch evaluation corresponding to the stable semantics maps positive loops to false.

A prototypical domain to apply knowledge representation and reasoning in is legal reasoning. I am particularly interested in an interactive decision enactment system, which makes legal decisions – such as determining how many taxes someone should pay – with interaction from the user. This interaction is in the form of the user supplying partial information.

In legal systems, a correct conclusion without a justification is frequently useless as you want to guarantee compliance with the law. My research will mainly be focussing on legal decisions that do not leave too much room for interpretations. Various types of public administration fall into this category.

In order to build such a system, a logic programming language with legal constructs is needed. Justification theory should then be used to define semantics for it since solutions of such a logic program need to be explained for it to be legally useful. The choice for justification theory is due to the fact that explanations are in the core of justification theory. However, many different approaches exist to provide explanations in answer set programming, see [9] for a survey, but justification theory additionally provides a unifying framework for different semantics.

This motivates why we need to integrate deontic (prohibited, obligatory, or permitted) and alethic (possibility, impossibility and necessity) modal operators into justification theory since they are key constructs in legal texts. This lan-

guage, and paired with that, extensions to justification theory will be the main driving factor throughout this extended abstract.

1.1 Crash course in justification theory

Before we dive into the actual material, a small crash course in justification theory is given. More detailed information can be found in [13] and [6].

Let $\mathcal{F}$ be any set of facts such that $\mathbf{t}, \mathbf{f}, \mathbf{u}$ and $\mathbf{i}$ are in $\mathcal{F}$. They are the logical truth values *true*, *false*, *unknown*, and *inconsistent*[3]. There is a negation $\sim$ on $\mathcal{F}$ such that $\sim x \neq x$ for non logical facts x in $\mathcal{F}$ and $\sim(\sim x) = x$ for all facts x. The negations of the logical facts are $\sim\mathbf{t} = \mathbf{f}$, $\sim\mathbf{u} = \mathbf{u}$ and $\sim\mathbf{i} = \mathbf{i}$. A *justification frame* is a fact space $\mathcal{F}$ equipped with a set R with elements of the form $x \leftarrow A$ where x is a non logical fact and A is a nonempty subset of $\mathcal{F}$. The elements of R are called *rules* with *head* x and *body* A. Each fact that occurs in the head of a rule is a *defined* fact. All other facts are called *open*.

Example 1. In this example, we build a justification frame to express the transitive closure of a graph. So let V be a set of nodes. Define the open elements to be the set of elements $\mathrm{Edge}(v, w)$ and $\sim\mathrm{Edge}(v, w)$ with $v, w \in V$ and the defined elements to be the set of elements $\mathrm{Path}(v, w)$ and $\sim\mathrm{Path}(v, w)$ with $v, w \in V$. The facts encoding the edges of a graph are open. This means that they can freely change and thus act as parameters, whereas the defined facts are constrained by the rules seen below. We define the rules[4] for $\mathrm{Path}(v, w)$:

$$\mathrm{Path}(v, w) \leftarrow \mathrm{Edge}(v, w);$$
$$\mathrm{Path}(v, w) \leftarrow \mathrm{Path}(v, x), \mathrm{Path}(x, w)$$

for all $v, w, x \in V$. This encodes that Path is the transitive closure of Edge.

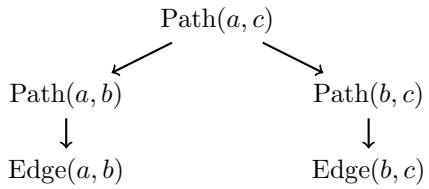
A *justification* over a justification frame is a subset of R containing at most one rule for each defined fact. We can also view a justification as a partial function from $\mathcal{F}$ to subsets of $\mathcal{F}$ by mapping a defined fact to the body of the rule in the justification if it exists. Differently, a justification can also be viewed as a graph, where we have arrows going from the head to elements in the body.

A justification is *locally complete* if for each defined fact occurring in the body of a rule in the justification, there is a rule with that fact as the head in the justification. If you view the justification as a graph, it means that no complete path ends in an open fact.

Example 2. Let $V = \{a, b, c\}$ in the setting of Example 1. Suppose that $\mathrm{Edge}(a, b)$ and $\mathrm{Edge}(b, c)$ hold. The following locally complete justification gives an explanation why $\mathrm{Path}(a, c)$ holds.

[3] They have the same structure as the four-valued logic of Belnap [1] with truth order $\mathbf{f} \leq_t \mathbf{u}, \mathbf{i} \leq_t \mathbf{t}$ and information order $\mathbf{u} \leq_k \mathbf{f}, \mathbf{t} \leq_k \mathbf{i}$.

[4] Normally in justification theory you add rules dual to these as well; rules that define $\sim\mathrm{Path}(v, w)$ and are compatible with the rules for $\mathrm{Path}(v, w)$.



A *branch* in a justification frame is either a finite sequence of defined facts together with an open fact; or an infinite sequence of defined facts. In Example 2, the following is a branch of the illustrated justification

$$\mathrm{Path}(a, c) \to \mathrm{Path}(a, b) \to \mathrm{Edge}(a, b).$$

A *branch evaluation* is then a mapping from branches to facts. The *well-founded branch evaluation* for example maps finite branches to its open fact and infinite branches to **t** if it has a negative tail, to **f** if it has a positive tail, and to **u** otherwise. The *stable branch evaluation* maps to the first element that has a different sign as the first element if it exists, otherwise it maps positive loops to **f** and negative loops to **t** and finite branches of the same sign to the last element.

Example 3. Take the following logic program

$$p \leftarrow {\sim}p$$

The only locally complete justification for p is given below

The well-founded branch evaluation maps the branch $p \to {\sim}p \to \cdots$ to **u**, while the stable branch evaluation maps it to ${\sim}p$.

A fact is *supported* by a locally complete justification in a subset $I \subseteq \mathcal{F}$ if each branch in the justification starting with that fact is mapped to I under the branch evaluation. A fact is *supported* in I if there is some locally complete justification supporting that fact in I.

Example 4. Let the setting be the same as in Example 2. Take the set $I = \{\mathrm{Edge}(a, b), \mathrm{Edge}(b, c)\}$. The fact $\mathrm{Path}(a, c)$ is supported in I under the Well-founded branch evaluation, a justification supporting $\mathrm{Path}(a, c)$ is already given in Example 2.

The *support operator* maps a subset I to the facts it supports. Intuitively, you can view it in the following way: I is the set of what you already know, and the support operator maps I to the facts that can be derived when the facts in I holds. To make the support operator iterable, we just add the elements of I back to the output of the support operator. This gives us the *extended support operator*. The (least) fixed points hereof are used to define justification semantics. In [6], it is proven that the justification semantics of the stable and well-founded branch evaluations match their equally-named logic programming semantics.

2 Central objective

As mentioned in the background, the central objective is to research extensions justification theory in several directions, to ultimately support legal decision systems.

2.1 First-order justifications

One limitation of the current state of justification theory is that it only deals with *ground* rules. This limitation is noticeable in lazy grounding techniques since ground justifications are much larger than first-order justifications, see e.g., [2]. Therefore, we see it fit to extend the theory to a first-order formalism. Keeping track of variables in loops makes it challenging to devise branch evaluations corresponding to the stable and well-founded semantics. In [12], the authors compute stable models without grounding with a goal-directed method. They use techniques that resemble ones in justification theory, but they work in a first-order setting. Therefore this work provides a base that we can build on.

2.2 Extending logic programing

In [6] it is mentioned that nesting of justification frames can be used to define semantics for logic programming with aggregates. One research question to tackle is whether this idea can be generalized to add arbitrary new logical constructs to logic programming. In the introduction, we discussed the use for modal operators such as alethic and deontic modalities. Therefore extending logic programming with these modalities is of particular interest to me. Some work has been done already in this area, most remarkably MOLOG, a system that extends Prolog with modal logic, see [3].

2.3 Nondefective justification frameworks

One advantage of justification theory is that both positive and negative facts can occur in the head of a rule. However this brings a caveat with it: a fact and its negation should not be supported simultaneously; if for example p and $\sim p$ are supported, then there should be inconsistencies in the rules. If this is not the case, we call the branch evaluation *defective*. Stated differently, a branch evaluation is non-defective if the following holds: if the rules are consistent, then for each non-logical fact p, we have that at most one of p and $\sim p$ is supported.

In recent work [13], we proved that this is the case for the well-founded, stable, Kripke-Kleene and Clarke completion branch evaluations. The proofs for these results are not easily generalisable to other branch evaluations. If we extend justification theory to a first-order setting or extend it with modal operators, we still need to prove that the corresponding branch evaluations are not defective. Therefore, we want to investigate which properties a branch evaluation needs to be non-defective to obtain general classes of non-defective branch evaluations.

2.4 Possible practical applications

As mentioned in the introduction, the research stipulated here can be used in more practical applications. A type of application that interests me, in particular, is an interactive decision enactment system. This is a system that makes a legal decision, such as granting an environmental permit, with interaction from the user. The interaction is in the form of the user providing partial information either from input or output.

Central to such a decision system is a modelling of a particular legislative text into a logical format. With this modelling, one can perform various kinds of reasoning. An important example of a type of reasoning in a legal context is providing explanations; explaining why the system has inferred some information. Since justifications can be seen as explanations, it is natural to consider a system based on justification theory. However, the underlying framework should be extended in order to support logical constructs present in legislation, e.g., deontic and alethic operators.

In such a system it often occurs that if you are in a particular partial configuration of the decision, some information is not needed any more, the information becomes *irrelevant*. The system should then only query relevant information to improve usability. In [8], we developed a notion of relevance in terms of justification theory.

Since both explanations and relevance can be defined with justification theory, we believe that justification theory is a good candidate for the back-end for an interactive decision enactment system.

More practically, I plan to work on Belgian law regarding termination of employee contracts. An example decision could be deciding how much compensation the employee receives.

2.5 Expressing new semantics

While many semantics of logic programming are captured by justification theory, some others cannot be expressed yet. For example, how do we express the ultimate stable and well-founded semantics [7]? Ultimate stable semantics is used to define semantics for logic programs with aggregates, see [7].

One idea is to alter the rules of the justification frame and use the support operator on this revised justification frame to define models by means of fixed points. If this approach is suitably general, it allows us to define ultimate semantics for various other formalisms.

Another idea is to devise a new branch evaluation for ultimate stable semantics and ultimate well-founded semantics. Several questions arise: how does the ultimate branch evaluations relate to the original branch evaluations? Can we define an 'ultimate' operator on branch evaluations that maps a branch evaluation to its corresponding ultimate branch evaluation?

When new semantics are defined they are often given terms of fixed points of a particular operator. Since justification semantics are also given in terms of fixed points of an operator associated with a branch evaluation, we can wonder

if there is a systematic way to determine the branch evaluation corresponding to the original semantics?

References

1. Belnap, N.D.: A useful four-valued logic. In: Dunn, J.M., Epstein, G. (eds.) Modern Uses of Multiple-Valued Logic, pp. 8–37. Reidel, Dordrecht (1977), invited papers from the Fifth International Symposium on Multiple-Valued Logic, held at Indiana University, Bloomington, Indiana, May 13-16, 1975
2. Bogaerts, B., Weinzierl, A.: Exploiting justifications for lazy grounding of answer set programs. In: Lang, J. (ed.) Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden. pp. 1737–1745. ijcai.org (2018). https://doi.org/10.24963/ijcai.2018/240, https://doi.org/10.24963/ijcai.2018/240
3. del Cerro, L.F.: MOLOG: A system that extends PROLOG with modal logic. New Generation Comput. **4**(1), 35–50 (1986). https://doi.org/10.1007/BF03037381, https://doi.org/10.1007/BF03037381
4. De Cat, B., Bogaerts, B., Devriendt, J., Denecker, M.: Model expansion in the presence of function symbols using constraint programming. In: 2013 IEEE 25th International Conference on Tools with Artificial Intelligence, Herndon, VA, USA, November 4-6, 2013. pp. 1068–1075. IEEE Computer Society (2013). https://doi.org/10.1109/ICTAI.2013.159, http://dx.doi.org/10.1109/ICTAI.2013.159
5. De Cat, B., Denecker, M., Bruynooghe, M., Stuckey, P.J.: Lazy model expansion: Interleaving grounding with search. J. Artif. Intell. Res. (JAIR) **52**, 235–286 (2015). https://doi.org/10.1613/jair.4591, http://dx.doi.org/10.1613/jair.4591
6. Denecker, M., Brewka, G., Strass, H.: A formal theory of justifications. In: Calimeri, F., Ianni, G., Truszczyński, M. (eds.) Logic Programming and Nonmonotonic Reasoning - 13th International Conference, LPNMR 2015, Lexington, KY, USA, September 27-30, 2015. Proceedings. Lecture Notes in Computer Science, vol. 9345, pp. 250–264. Springer (2015). https://doi.org/10.1007/978-3-319-23264-5_22, http://dx.doi.org/10.1007/978-3-319-23264-5_22
7. Denecker, M., Pelov, N., Bruynooghe, M.: Ultimate well-founded and stable semantics for logic programs with aggregates. In: Logic Programming, 17th International Conference, ICLP 2001, Paphos, Cyprus, November 26 - December 1, 2001, Proceedings. pp. 212–226 (2001)
8. Deryck, M., Devriendt, J., Marynissen, S., Vennekens, J.: Legislation in the knowledge base paradigm: interactive decision enactment for registration duties. In: Proceedings of the 13th IEEE Conference on Semantic Computing (ICSC), Newport Beach, California, USA, Jan. 30 - Feb. 1, 2019. pp. 174–177. IEEE (2019)
9. Fandinno, J., Schulz, C.: Answering the "why" in answer set programming - A survey of explanation approaches. TPLP **19**(2), 114–203 (2019). https://doi.org/10.1017/S1471068418000534, https://doi.org/10.1017/S1471068418000534
10. Gebser, M., Kaufmann, B., Schaub, T.: Conflict-driven answer set solving: From theory to practice. Artif. Intell. **187**, 52–89 (2012)
11. Jansen, J., Bogaerts, B., Devriendt, J., Janssens, G., Denecker, M.: Relevance for SAT(ID). In: Kambhampati, S. (ed.) Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New

York, NY, USA, 9-15 July 2016. pp. 596–603. IJCAI/AAAI Press (2016), http://www.ijcai.org/Abstract/16/091
12. Marple, K., Salazar, E., Gupta, G.: Computing stable models of normal logic programs without grounding. CoRR **abs/1709.00501** (2017), http://arxiv.org/abs/1709.00501
13. Marynissen, S., Passchyn, N., Bogaerts, B., Denecker, M.: Consistency in justification theory. In: Proceedings of 17th International Workshop on Non-Monotonic Reasoning (NMR 2018), Tempe, Arizona, USA, Oct. 27-29, 2018. pp. 41–52. AAAI Press 2018 (2018), http://www4.uma.pt/nmr2018/NMR2018Proceedings.pdf