Joint Proceedings of the 2nd Workshop on Natural Language Processing and Automated Reasoning, and the 2nd International Workshop on Learning and Nonmonotonic Reasoning at LPNMR 2015

Editors:
Marcello Balduccini[1], Alessandra Mileo[2], Ekaterina Ovchinnikova[3], Alessandra Russo[4], and Peter Schüller[5]

[1] Drexel University, USA
[2] INSIGHT Centre for Data Analytics, NUI Galway, Ireland
[3] KIT, Karlsruhe & ICT, Uni Heidelberg, Germany
[4] Dept. of Computing, Imperial College London, UK
[5] Marmara University, Turkey

Preface

This volume contains the papers presented at the joint NLPAR 2015 and LNMR 2015 workshops: 2nd Workshop on Natural Language Processing and Automated Reasoning, and 2nd International Workshop on Learning and Nonmonotonic Reasoning, held on September 27, 2015, in Lexington, USA.

The NLPAR Workshop received 3 submissions and the LNMR workshop received 2 submissions from different international institutions and research communities. Each submission was reviewed by 3 program committee members. The committee decided to accept 3 papers. The program also includes 2 invited talks.

The organizing committee wants to thank the Workshop Chair of LPNMR, Yuliya Lierler, and the Chairs of LPNMR, Victor Marek, Giovambattista Ianni, and Mirek Truszczyński, for their support in embedding this workshop into the LPNMR conference organization.

This workshop was managed using EasyChair.

Marcello Balduccini
Alessandra Mileo
Ekaterina Ovchinnikova
Alessandra Russo
Peter Schüller

Table of Contents

Two Advances in the Implementations of Extended Syllogistic Logics

Jason Hemann, Cameron Swords, and Lawrence S. Moss
{jhemann, cswords, lmoss}@indiana.edu

Indiana University, Bloomington

Abstract. Natural logics are of interest to both logicians and members of the natural-language research community. They provide a means of precisely reasoning about aspects of natural language in a way that is computationally tractable, and akin to the process by which humans reason in ordinary language. This paper takes as a target a reasonably-small logic which support reasoning about "All", "Some", negated nouns, relative clauses, and the "more X than Y" operation of cardinality comparison. The importance of this logic is that it goes beyond first-order logic, hence one cannot use off-the-shelf tools. This paper contains two contributions to the implementation of this logic and others. First, it mentions an implementation in Sage. The program builds proofs and counter-models by one and the same algorithm. That is, the failure to build a proof provides the data for a counter-model in an automatic way. This abstract does not go into details on the algorithm, but a talk on this includes a demo of the Sage program. Second, in a very different direction, we mention *declarative implementations* of a different logic in this family, done in the miniKanren language. These implementations provide users with automated proof search, theorem generation, and proof checking, and are designed to facilitate reuse in implementing other natural logics.

1 Introduction

Logical syllogisms—arguments with deductive reasoning—have been the object of study since at least Aristotle. More recently, these logical syllogisms become the core of a series of *natural logics* [14], which aim to mirror the style of deductions people employ in everyday reasoning.

Though well-studied, there has been little work toward developing automated tools for proof searches in natural logics. In the applications work that *has* been undertaken [22,19,18,17,4,12,27,25], the implementations themselves have not been the object of study; implementers have been content to use imperative implementations, or rely on SAT solvers or tools like Prover9 and Sage. These can provide powerful, performant tools for working with these logics. But, they somewhat obscure the direct nature of the reasoning employed. A high-level, declarative implementation of these logics, on the other hand, preserves and highlights the direct reasoning of these deduction systems in their implementations. This declarative approach also provides to be extensible and well-suited for rapid prototyping.

We utilize the declarative language miniKanren to demonstrate this encoding process for several logics, including a new logic for cardinality comparison of atoms, from their proof-tree derivation rules. For each logic, we produce for each a single tool that can be used for proof instantiation, proof derivation, and automated theorem search from a list of premises. The full code may be found at `http://github.com/jasonhemann/natlogic`. This paper proceeds as follows:

- Section 2 discusses the logic with cardinality comparison, and it shows examples of the Sage implementation.
- Section 3 serves as a brief primer to the miniKanren language and the cKanren implementation, embedded in Racket. It also describes the basic strategy to encode natural logics as miniKanren programs, including premise representation, proof construction, and user invocation. We demonstrate this approach by encoding $\mathcal{A}$, the logic of 'All", in miniKanren.
- Section 4 describes a miniKanren implementation of a logic with cardinality comparison.
- Section 5 discusses related work and concludes and describes potential future work.

2 A Logic for Cardinality Comparison

We begin by introducing a logic for cardinality comparison on top of the basic syllogistic logic, taken from [20]. Consider the following argument:

$$\frac{\begin{array}{l}\textsf{There are more students than professors at the party}\\ \textsf{There are more professors than deans at the party}\end{array}}{\textsf{There are more students than deans at the party}} \qquad (1)$$

The conclusion follows from the premises. The intuition is that the transitivity of more ... than ... is a basic feature of human reasoning, on a par with the transitivity of all ... are ... that we see in the syllogistic rule (BARBARA). We do not wish to formalize the argument in (1) by translating it into another logic (for example, logical systems which incorporate natural numbers); the point is that the general logical principles of the target systems are likely to be much more complicated than necessary for this task.

Let us widen the discussion a little. In addition to more ... than ..., we also find in language the weaker assertion there are at least as many ... as .... Here is another argument which we take to be valid:

$$\frac{\begin{array}{l}\textsf{There are at least as many rabbits as deer}\\ \textsf{There are more deer than goats}\end{array}}{\textsf{There are more rabbits than goats}} \qquad (2)$$

$$\frac{}{\forall(p,p)}\ (\textsc{axiom}) \qquad \frac{\forall(n,p)\quad \forall(p,q)}{\forall(n,q)}\ (\textsc{barbara})$$

$$\frac{\exists(p,q)}{\exists(p,p)}\ (\textsc{some}) \qquad \frac{\exists(q,p)}{\exists(p,q)}\ (\textsc{conversion})$$

$$\frac{\exists(p,n)\quad \forall(n,q)}{\exists(p,q)}\ (\textsc{darii}) \qquad \frac{\forall(p,q)\quad \exists^{\geq}(p,q)}{\forall(q,p)}\ (\textsc{card-mix})$$

$$\frac{\forall(p,q)}{\exists^{\geq}(q,p)}\ (\textsc{subset-size}) \qquad \frac{\exists^{\geq}(n,p)\quad \exists^{\geq}(p,q)}{\exists^{\geq}(n,q)}\ (\textsc{card-trans})$$

$$\frac{\exists(p,p)\quad \exists^{\geq}(q,p)}{\exists(q,q)}\ (\textsc{card-}\exists) \qquad \frac{\exists^{>}(p,q)}{\exists^{\geq}(p,q)}\ (\textsc{more-at least})$$

$$\frac{\exists^{>}(n,p)\quad \exists^{\geq}(p,q)}{\exists^{>}(n,q)}\ (\textsc{more-left}) \qquad \frac{\exists^{\geq}(n,p)\quad \exists^{>}(p,q)}{\exists^{>}(n,q)}\ (\textsc{more-right})$$

$$\frac{\exists^{\geq}(p,q)\quad \exists^{>}(q,p)}{\phi}\ (\mathrm{X})$$

$$\frac{\forall(p,\overline{p})}{\forall(p,q)}\ (\textsc{zero}) \qquad \frac{\forall(\overline{p},p)}{\forall(q,p)}\ (\textsc{one})$$

$$\frac{\forall(q,p)\quad \exists(p,\overline{q})}{\exists^{>}(p,q)}\ (\textsc{more}) \qquad \frac{\exists^{>}(p,q)}{\exists(p,\overline{q})}\ (\textsc{more-some})$$

$$\frac{\exists^{>}(q,p)}{\exists^{>}(\overline{p},\overline{q})}\ (\textsc{more-anti}) \qquad \frac{\forall(p,q)}{\forall(\overline{q},\overline{p})}\ (\textsc{anti})$$

$$\frac{\exists^{\geq}(p,q)}{\exists^{\geq}(\overline{q},\overline{p})}\ (\textsc{card-anti}) \qquad \frac{\exists(p,p)\quad \exists^{\geq}(q,\overline{q})}{\exists(q,q)}\ (\textsc{int})$$

$$\frac{\exists^{\geq}(p,\overline{p})\quad \exists^{\geq}(\overline{q},q)}{\exists^{\geq}(p,q)}\ (\textsc{half}) \qquad \frac{\exists^{>}(p,\overline{p})\quad \exists^{\geq}(\overline{q},q)}{\exists^{>}(p,q)}\ (\textsc{strict half})$$

$$\frac{\exists^{\geq}(p,\overline{p})\quad \exists^{\geq}(q,\overline{q})\quad \exists(\overline{p},\overline{q})}{\exists(p,q)}\ (\textsc{maj})$$

Fig. 1. Rules for the cardinality logic. The rules for a smaller system, which lacks complemented variables, are found above the line.

And here is an argument of a different character:

$$\frac{\begin{array}{l}\textsf{All violas are stringed instruments}\\ \textsf{There are at least as many violas as stringed instruments}\end{array}}{\textsf{All stringed instruments are violas}} \tag{3}$$

A moment's thought will convince the reader that this is valid, provided that we are speaking of *finite* situations. In this paper we restrict attention to finite universes, in order to obtain a logical system that we think of greater "human interest" than the weaker logic that would result if we allowed infinite structures and thus denied the validity of (3).

Finally, we make our logical language more expressive by allowing *complementation of nouns*. Here are some examples:

$$\begin{array}{c}\dfrac{\textsf{There are at least as many } x \textsf{ as } y}{\textsf{There are at least as many non-}y \textsf{ as non-}x}\\[2em] \dfrac{\begin{array}{l}\textsf{There are at least as many } x \textsf{ as non-}x\\ \textsf{There are at least as many } y \textsf{ as non-}y\end{array}}{\textsf{There are at least as many } x \textsf{ as non-}y}\end{array} \tag{4}$$

The first example just above shows an inference whose soundness depends on the fact that we are looking at a finite universe. The second uses a property of "half": if the universe has N objects, the premises tell us that the xs are at least $\frac{N}{2}$ in number. The ys number at least $\frac{N}{2}$, and so the non-ys number at most $\frac{N}{2}$. Thus the xs number at least as much as the non-ys. The fact that we can do all of this with cardinality comparison and complement makes this work interesting and non-trivial.

The main result in [20] is a sound and complete logical system whose sentences are of the form All x are y, Some x are y, There are at least as many x as y, and There are more x than y. Moreover, th logic does not involve translating the cardinality assertions into any other language. The proof system is sound and strongly complete: for a finite set $\Gamma \cup \{\phi\}$ of sentences, ϕ is true in every model of Γ if and only if there is a derivation of ϕ from Γ. This paper does not discuss the completeness result at all but rather presents the implementation.

Formal System. Figure 1 presents the rules of the system in natural-deduction format. The logic has sentences of the following forms:

- $\forall(p, q)$ read as "all p are q",
- $\exists(p, q)$ read as "Some p are q",
- $\exists^{\geq}(p, q)$ read as "there are at least as many p as q",
- $\exists^{>}(p, q)$ read as "there are more p than q"

There are no connectives, and the overline symbol ($\overline{p}$) on the variables is for set complement. This logic induces a notion of models and precise definitions of model satisfaction such that we may define what it means for a (finite) set of sentences to semantically imply another sentence.

Implementation. On the other hand, this paper is about the implementation. The logical consequence has been implemented in Sage, and the implementation is currently available on `https://cloud.sagemath.com`. (That is, it can be shared.) The consequence relation may be computed in polynomial time. This should be a little surprising, since the cardinality comparison machinery cannot be expressed in first-order logic.

Example 1. One may enter:

```
 assumptions= ['All non-a are b',
              'There are more c than non-b',
              'There are more non-c than non-b',
              'There are at least as many non-d as d',
              'There are at least as many c as non-c',
              'There are at least as many non-d as non-a']
conclusion = 'All a are non-c'
follows(assumptions,conclusion)
```

The last line indicates that we are asking if a given conclusion follows from a given list of six assumptions. Then the program returns, telling us that the conclusion does not follow. And it produces a *counter-model*, a model where all of the assumptions are true and the conclusion false.

```
Here is a counter-model.
We take the universe of the model to be {0, 1, 2, 3, 4, 5}

  noun   semantics      complement
+------+--------------+--------------------+
  a      {2, 3}         {0, 1, 4, 5}
  b      {0, 1, 4, 5}   {2, 3}
  c      {0, 2, 3}      {1, 4, 5}
  d      {}             {0, 1, 2, 3, 4, 5}
```

So it gives the semantics of `a`, `b`, `c`, and `d` as subsets of $\{0, \ldots, 5\}$. Notice that the assumptions are true in the model, but the conclusion is false. In the cases that the conclusion did follow, the system would output a proof in our system.

Example 2. Here is an example of a derivation found by our implementation. We ask whether the putative conclusion below really follows:

$$\frac{\begin{array}{l}\textsf{All non-x are x}\\ \textsf{Some non-y are z}\end{array}}{\textsf{There are more x than y}}$$

The program returns the following result when we provide it the assumptions listed above, asking for a profe that there are more x than y:

```
1   All             non−x   are    x        Assumption
2   All             y       are    x        One   1
3   All             non−x   are    x        Assumption
4   All             non−y   are    x        One   3
5   Some            non−y   are    z        Assumption
6   Some            non−y   are    non−y    Some  5
7   Some            non−y   are    x        Darii   4   6
8   Some            x       are    non−y    Conversion   7
9   There are more  x       than   y        More   2   8
```

While the proof is displayed as a list rather than a tree, it is merely a cosmetic difference.

The advantage of working with a syllogistic system formulated using *ex falso quodlibet* rather than *reductio ad absurdum* is that the proof search and the counter-model generation are closely related. In a sense, they are both results of the same algorithm. Moreover, the algorithm is efficient. That is, the question of whether a sentence follows from a list of assumptions is in polynomial time.

Unfortunately, the implementation is obscuring: it is over 1500 lines of Sage and ultimately relies on iteratively constructing *all possible derivations* from a given set of assumptions. Moreover, this implementation is customized toward dealing with the logic of relative cardinalities. These features are hardly ideal when experimenting with a new logic.

To this end, we now turn our focus to miniKanren, a declarative programming language embedded in Racket, to demonstrate how it can be used to develop proof searches for natural logics. Unlike our Sage work, the miniKanren implementation of this cardinality logic is concise, clear, and extensible. It is, however, unable to generate counter-models and takes super-polynomial time: the miniKanren approach is good for experimentation, but has sub-par performance.

3 Implementations in miniKanren

Our previous section discussed a Sage implementation of a single large logic with distinct advantages in disadvantages. In its favor, the algorithm works in polynomial time and includes counter-model generation along with proof search. Unfortunately, the algorithm is highly specialized toward the logics and difficult to explain (and thus eschewed in our presentation). In a different direction, we present a generic way to build *declarative implementations* of syllogistic logics. The key point is the genericity of the work: it is straightforward to construct and experiment with proof searches for these logics in a declarative language. On the other hand, these constructions are not as closely related to counter-model search and the resultant encodings are less efficient than custom-constructed algorithms.

3.1 miniKanren: A Brief Introduction

miniKanren is a family of embedded, domain-specific relational (logic) programming languages [3,5,6,2,10,9]. Implementations come with a variety of constraints, the foremost of which is `==`, an equality constraint implemented with syntactic, first order unification. For this presentation, we use cKanren, an implementation of miniKanren embedded in Racket [7]. The cKanren implementation provides programmers with access to the entirety of the host language when writing miniKanren programs[1] and the ability to define their own constraints.

The `run` Interface. The primary interface to miniKanren is `run`, which takes the *maximal* number of answers desired, an "output" variable—a variable with respect to which the answer should be presented—and a sequence of goal expressions to achieve. Consider the following miniKanren program execution:

```
> (run 1 (q) (== q 3))
'(3)
```

Here, the `1` indicates we request at most one answer for the variable `q`. The only constraint is the equality of `q` with `3`. The output is always presented as a list of results; this is the list contains only one result for `q`, the value `3`. Consider this second execution:

```
> (run 1 (q) (== 3 3))
'(_.0)
```

The list of results again contains only one element, this time `_.0`. The final substitution for this program has no information regarding `q`, it is a *fresh variable*. In the presentation of the answer, distinct fresh variables are written `_.`n, where n is an index beginning at 0.

Getting `fresh` Variables. It is often useful to introduce auxiliary logic variables as a part of writing a miniKanren program. In the example below, we wish to assert the query variable is a pair; we introduce new variables `a` and `d` and use them in a constraint:

```
> (run 1 (q) (fresh (a b) (== q `(,a . ,b))))
'((_.0 . _.1))
```

The `fresh` operator takes a list of identifiers and a sequence of goal expressions over which new variables are scoped. In this case, they are scoped over a constraint equating `q` with a pair whose first element is the variable `a` and whose second is the variable `b`. We rely on the host language's term constructors to build miniKanren terms, destructuring must be performed with `==`. New variables are lexically scoped, so inner bindings shadows outer ones.

[1] Except vectors, which are used in the implementation.

Using `conde` for Non-deterministic Computation. The `conde` operator implements a complete search (whose details are unimportant here) that allows us to simulate a form of nondeterministic choice. It takes any number of clauses (lists of goal expressions) and operates as though each clause were attempted independently.

```
> (run 3 (q) (conde
              ((fresh (a b) (== q `(,a . ,b))))
              ((== q 6))))
'(6 (_.0 . _.1))
```

We request 3 results, but receive only two: one from each `conde` clause. miniKanren interleaves the search for results, and we are in general not guaranteed to receive the results in the order of their `conde` clauses.

Disequality Constraints. The miniKanren operator `=/=` implements disequality constraints. Placing a disequality constraint on two terms already identical in the current substitution causes failure, and if `u` and `v` are under a disequality constraint, then a substitution extension that forces `u` and `v` to be syntactically identical will also cause failure. Like the substitution, disequality constraints are carried as part of the state and are indicated in the output:

```
> (run 1 (q) (=/= q 3))
'((_.0 (=/= ((_.0 3)))))
```

The variable `q` still has no binding in the ultimate substitution, and so it is again presented as `_.0`, but we also mandate that `_.0` not be 3. Our disequality constraints, like the `dif/2` of various Prologs, fail only when their arguments are identical relative to the current substitution.

User-defined Constraints. We demonstrate an example of a user-defined cKanren constraint below. We provide a name, and specify its criteria for satisfaction and its interactions with other constraints. As part of the implementations we provide a suite of pre-built constraints for defining these and other natural logics.

The `un-atom` constraint mandates that the term be a unary atom, which for our purposes means a plural noun (e.g. "logicians"). We represent them as symbols, and we require they not overlap with binary atoms (transitive verbs).

```
(define-attribute un-atom
  #:satisfied-when symbol?
  #:incompatible-attributes (number bin-literal bin-atom))
```

Adding constraints for atoms, literals, negated literals, etc., makes the resulting answers more legible and also groups together multiple answers by collapsing the search space.

3.2 Putting Things Together

A miniKanren program attempts to satisfy a number of *goals* in a given *state*, which either *succeed*, returning a *stream* of one or more achieving states, or *fail*, yielding an empty stream. Because cKanren is embedded in Racket, miniKanren programmers have access to the entirety of Racket when writing miniKanren programs. As a result, relations can be defined globally and then invoked elsewhere, as in the following example:

```
> (define (membero x l)
    (fresh (a d)
      (== l `(,a . ,d))
      (conde
        ((== x a))
        ((=/= x a) (membero x d)))))
```

We globally define the binary relation `membero`, which holds when `x` is an elements of a list `l`. In the program below, we demand `q` be such a list containing `'x`, and we request three such elements.

```
> (run 3 (q) (membero 'x q))
'((x . _.0)
  ((_.0 x . _.1) (=/= ((_.0 x))))
  ((_.0 _.1 x . _.2) (=/= ((_.0 x)) ((_.1 x)))))
```

Furthermore, we can use Racket's macro system to extend miniKanren with additional syntactic operations, such as `matche`, a pattern matcher that will perform automatic fresh variable creation [13]. For example, consider the following two definitions of relational append:

```
(define (appendo ls1 ls2 lout)
  (conde
    ((== ls1 '())
    (== ls2 lout))
    ((fresh (a d r)
      (== ls1 `(,a . ,d))
      (== lout `(,a . ,r))
      (appendo d ls2 r)))))
```

```
(define (appendo ls1 ls2 lout)
  (matche ls1
    (()
     (== ls2 lout))
    ((,a . ,d)
     (fresh (r)
       (== lout `(,a . ,r))
       (appendo d ls2 r)))))
```

The left one creates a number of fresh variables and performs unification against `ls1` at each step. The right one performs the same operations with the pattern matching tool `matche`, dispatching on the shape of `ls1`. This approach allows us to avoid creating a number of additional variables and elide the unifications against `ls1` in the program[2]. This style of match-and-dispatch will prove invaluable in rapidly constructing logical proof search.

[2] These equations and fresh variables are created during macro expansion.

3.3 A System for Logical Encoding

With this basic understanding of natural logics and miniKanren, we now proceed with encoding natural logics in a generic and extensible way. We begin with $\mathcal{A}$, the logic of "All" relations [17,19,22,8], to demonstrate the general encoding process. $\mathcal{A}$ uses three judgement rules: environmental lookup, (AXIOM), and (BARBARA) from Figure 1. We use Γ to represent the set of premises.

These rules indicate that every object, or *unary atom*, p, is reflexively self-contained, and the containment relation is transitive. To encode these rules in miniKanren, we must build a relation that takes as arguments a theorem `φ` to prove, some environment of premises, `Γ`, and, because we are writing a relation, some proof tree output `proof`. The resultant procedure, presented in Figure 2, uses miniKanren's `==`, `conde`, `matche`, and `fresh` as well as the aforementioned parts of the host language.

```
1  (define (A φ Γ proof)
2    (matche φ
3      [(∀ ,a ,a) (== φ proof)] ;; Axiom
4      [,x (membero x Γ) (== proof `(,x in−Γ))]  ;; Lookup
5      [(∀ ,n ,q) ;; Barbara
6       (fresh (p prim1 proof1 prim2 proof2)
7         (== `((,proof1 ,proof2) => ,φ) proof)
8         (== prim1 `(∀ ,n ,p))
9         (== prim2 `(∀ ,p ,q))
10        (A prim1 Γ proof1)
11        (A prim2 Γ proof2))]))
```

Fig. 2. A miniKanren implementation for $\mathcal{A}$.

Our implementation operates over not-quite-English: like McAllester and Givan [15], we find it convenient to encode the premises provided as lists. We encode "All" sentences as `(∀ p q)` instead of McAllester and Givan's `(All p q)` structure: Racket supports unicode so we can more closely match the format of the logical rules.

This procedure is the entire encoding of $\mathcal{A}$. We begin by matching against the input `φ` and proceeding with three possibilities (one for each rule):

- The first, at line 3, asks if `φ` will unify with `(∀ ,a ,a)`—if we are stating that "All a are a." In this case, the proof follows trivially (by (AXIOM)), and thus we unify the statement with the proof tree output.
- The second, at line 4, matches generically against any `φ` and then checks if that `φ` is a member of `Γ`. If it is, we unify the proof tree with a list denoting the entailment.
- The third, on lines 6–11, encodes the transitivity rule (BARBARA) of $\mathcal{A}$. We introduce five fresh variables:

- `p`, the intermediary atom in the term
- `prim1` and `prim2`, which represent `(∀ ,n ,p)` and `(∀ ,p ,q)` respectively
- `proof1` and `proof2`, which indicate the proof terms for `(∀ ,n ,p)` and `(∀ ,p ,q)` respectively

Finally, we invoke `A` to recursively build proofs for `(∀ ,n ,p)` and `(∀ ,p ,q)`, passing in the appropriate proof variables in each case.

4 Cardinality Logic in miniKanren

With these tools in mind, we implement the cardinality object from Figure 1 in miniKanren. The first half (above the line in Figure 1) is given in Figure 3.

Similar to the preceding examples, the Racket function `card` implements a miniKanren relation that describes when that relationship holds between its inputs, and each `matche` clause of the `card` relation corresponds to a rule of Figure 1.

For larger logics such `card`, the repetition inherent in specifying the rules of the logic may become tedious: each two-premise recursion mirrors our implementation of (BARBARA) in Figure 2, and the one-premise rules follow similarly. We use Racket's macro system to once again simplify our task, creating two additional syntactic forms that construct the appropriate miniKanren terms. These new syntactic forms, `single-prim-term` and `double-prim-term` in Figure 3, take the logic's name, the environment, the term(s) that is required to hold in order for `φ` to hold, and any auxillary variables required, and use them to construct the fresh variable creation, unifications, and recursions necessary to implement the rule. For example, consider our equivalent implementations of (BARBARA) side-by-side:

```
[(∀ ,n ,q) ;; Barbara
 (fresh (p prim1 proof1 prim2 proof2)
   (== `((,proof1 ,proof2) => ,φ) proof)
   (== prim1 `(∀ ,n ,p))
   (== prim2 `(∀ ,p ,q))
   (A prim1 Γ proof1)
   (A prim2 Γ proof2))]
```

```
[(∀, n, q) ;; Barbara
 (double-prim-term
   card proof φ Γ
   `(∀ ,n ,p)
   `(∀ ,p ,q)
   p)]
```

Using these syntactic abstractions, our program is reduced to a series of pattern-matching clauses whose the left-hand sides are the translations of the conclusions of a judgment rule, and whose right-hand sides are invocations of `single-prim-term` or `double-prim-term`, encoding the antecedent or antecedents. This direct correspondence between the logical rules and the implementation facilitates rapid prototyping and quick experimentation when working with natural logics.

5 Conclusions and Next Steps

Interest in the history of syllogistic logics motivated the development of a great variety of tools (e.g., Glashof [11]). In particular, the work in Prolog has focused

```
(define-syntax double-prim-term
  (syntax-rules ()
     [(_ logic proof φ Γ e1 e2 vars ...)
      (fresh (vars ... prim1 prim2 proof1 proof2)
        (== `((,proof1 ,proof2) => ,φ) proof)
        (== prim1 e1)
        (== prim2 e2)
        (logic prim1 Γ proof1)
        (logic prim2 Γ proof2))]))

(define-syntax single-prim-term
  (syntax-rules ()
     [(_ logic proof φ Γ e1 vars ...)
      (fresh (vars ... prim1 proof1)
        (== `((,proof1) => ,φ) proof)
        (== prim1 e1)
        (logic prim1 Γ proof1))]))

(define (card φ Γ proof)
  (matche φ
    [(∀ ,a ,a) (== φ proof)] ;; Axiom
    [(∀, n, q)
     (double-prim-term card proof φ Γ `(∀ ,n ,p) `(∀ ,p ,q) p)] ;; Barbara
    [(∃ ,p ,p) ;; ∃
     (single-prim-term card proof φ Γ `(∃ ,p ,q) q)]
    [(∃ ,p ,q) ;; Conversion
     (single-prim-term card proof φ Γ `(∃ ,q ,p))]
    [(∃ ,p ,q) ;; Darii
     (double-prim-term card proof φ Γ `(∃ ,p ,n) `(∀ ,n ,q) n)]
    [(∀ ,q ,p) ;; Card-Mix
     (double-prim-term card proof φ Γ `(∀ ,p ,q) `(∃≥ ,p ,q))]
    [(∃≥ ,q ,p) ;; Subset-Size
     (single-prim-term card proof φ Γ `(∀ ,p ,q))]
    [(∃≥ ,n ,q) ;; Card-Trans
     (double-prim-term card proof φ Γ `(∃≥ ,n ,p) `(∃≥ ,p ,q) p)]
    [(∃ ,q ,q) ;; Card-E
     (double-prim-term card proof φ Γ `(∃ ,p ,p) `(∃≥ ,q ,p) p)]
    [(∃≥ ,p ,q) ;; More-At-Last
     (single-prim-term card proof φ Γ `(∃> ,p ,q))]
    [(∃> ,n ,q) ;; More-Left
     (double-prim-term card proof φ Γ `(∃> ,n ,p) `(∃≥ ,p ,q) p)]
    [(∃> ,n ,q) ;; More-Right
     (double-prim-term card proof φ Γ `(∃≥ ,n ,p) `(∃> ,p ,q) p)]
    [,x (membero x Γ) (== proof `(,x in-Γ))]  ;; Lookup
    [,x ;; X
     (double-prim-term card proof φ Γ `(∃≥ ,p ,q) `(∃≥ ,q ,p) p q)]))
```

Fig. 3. miniKanren implementation of the positive portion of the cardinality logic in Figure 1.

on natural language processing and the classical syllogistic logics (typically no further than $\mathcal{S}^{\dagger}$) [23,16,26,22,19,18,4,12,27,25]. There are a variety of such results, and it would be difficult to thoroughly catalog all of these systems here.

Our work with Sage shows that it is possible to do proof search and counter-model generation at the same time. The key point is that *reductio ad absurdum* is a derived rule, not a basic feature of the system. This is what is behind our polynomial-time algorithm.

The relational nature of miniKanren facilitates proof verification and proof search in the same implementation, and the generic style of implementation allows us to freely explore new extensions to the syntax and proof theory with little to no additional overhead. By default, miniKanren relies on a kind of breadth-first search strategy. Modifying these implementations, using techniques pioneered in rKanren [24], will allow the user to more finely tune the direction of the proof search. Additionally, future improvements in cKanren's set constraint architecture will likely enable an increase in both performance and clarity of our implementations.

But the most important next step in this line of work is to connect with the tableau system in [1] (based on [21]). Abzianidze's paper shows that computational systems based on natural logics can be combined with CCG parsers and other NLP tools in order to scale up work in this area. Indeed, he succeeds in handling RTE-like data. Nevertheless, his approach is based on tableaux, and ours is based on the complementary technique of formal proofs. So connecting the two approaches is the most important task on the road toward using natural logic in NLP.

References

1. Lasha Abzianidze. A tableau prover for natural logic and language. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. ACL, 2015.
2. Claire E Alvis, Jeremiah J Willcock, Kyle M Carter, William E Byrd, and Daniel P Friedman. cKanren: miniKanren with constraints. *Scheme and Functional Programming*, 2011.
3. William E. Bird. minikanren.org. `http://minikanren.org/`. Accessed 1/18/2014.
4. Patrick Blackburn and Johan Bos. *Representation and Inference for Natural Language: A First Course in Computational Semantics (Studies in Computational Linguistics)*. Center for the Study of Language and Information, 2005.
5. William E Byrd. *Relational programming in miniKanren: techniques, applications, and implementations*. PhD thesis, Indiana University, 2009.
6. William E Byrd, Eric Holk, and Daniel P Friedman. minikanren, live and untagged. *Scheme and Functional Programming*.
7. Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Design Inc., 2010. `http://racket-lang.org/tr1/`.
8. Nissim Francez and Roy Dyckhoff. Proof-theoretic semantics for a fragment of natural language. *Linguistics and Philosophy*, 33(6):447–477, 2011.
9. Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. *The Reasoned Schemer*. The MIT Press, July 2005.

10. Daniel P. Friedman and Oleg Kiselyov. A declarative application logic programming system, 2005.
11. Klaus Glashof. Computational aristotelian term logic, 2004. see http://webapp5.rrz.uni-hamburg.de/syllogism/aristotelianlogic/.
12. Nikolay Ivanov and Dimiter Vakarelov. A system of relational syllogistic incorporating full boolean reasoning. *Journal of Logic, Language, and Information*, 21(4):433–459, 2012.
13. Andrew W. Keep, Michael D. Adams, Lindsey Kuper, William E. Byrd, and Daniel P. Friedman. A pattern matcher for miniKanren or how to get into trouble with CPS macros. In *Scheme '09: Proceedings of the 2009 Scheme and Functional Programming Workshop*, number CPSLO-CSC-09-03 in California Polytechnic State University Technical Report, pages 37–45, 2009.
14. George Lakoff. Linguistics and natural logic. *Synthese*, 22:151–271, 1970.
15. David A. McAllester and Robert Givan. Natural language syntax and first-order inference. *Artificial Intelligence*, 56:1–20, 1992.
16. M McCord. Using slots and modifiers in logic grammars for natural language. *Artificial Intelligence*, 18(3):327–367, May 1982.
17. Lawrence S. Moss. Completeness theorems for syllogistic fragments. In F. Hamm and S. Kepser, editors, *Logics for Linguistic Structures*, pages 143–173. Mouton de Gruyter, 2008.
18. Lawrence S. Moss. Syllogistic logic with complements. In *Games, Norms and Reasons: Proceedings of the Second Indian Conference on Logic and its Applications*, page 19 pp. Springer Synthese Library Series, Mumbai, 2010.
19. Lawrence S. Moss. Notes on natural logics. unpublished ms., Indiana University, 2013.
20. Lawrence S. Moss. Syllogistic logic with cardinality comparisons. In Katalin Bimbo, editor, *J. Michael Dunn on Information Based Logics*, Outstanding Contributions to Logic. Springer-Verlag, to appear.
21. Reinhard Muskens. An analytic tableau system for natural logic. In Maria Aloni, Harald Bastiaanse, Tikitu de Jager, and Katrin Schulz, editors, *Logic, Language and Meaning*, volume 6042 of *Lecture Notes in Computer Science*, pages 104–113. Springer Berlin Heidelberg, 2010.
22. Ian Pratt-Hartmann and Lawrence S. Moss. Logics for the relational syllogistic. *Review of Symbolic Logic*, 2(4):647–683, 2009.
23. John F. Sowa. Conceptual graphs: Online course in knowledge representation using conceptual graphs. `http://cg.huminf.aau.dk/index.html`. Accessed 1/18/2014.
24. Cameron Swords and Daniel Friedman. rKanren: Guided search in miniKanren. *Scheme and Functional Programming*, 2013.
25. Jan van Eijck. Natural logic for natural language. In *Logic, Language, and Computation*, volume 4363 of *LNAI*, pages 216–230. Springer-Verlag, 2007.
26. Adrian Walker. *Knowledge Systems and Prolog: A Logical Approach to Expert Systems and Natural Language Processing*. Addison-Wesley, 1987.
27. Peter Yule and Buccleuch Place. A Prolog implementation of the method of Euler circles for syllogistic reasoning, 1996.

Answer Set Programming for Controlled Natural Language Processing

Rolf Schwitter and Stephen C Guy

Department of Computing
Macquarie University
Sydney 2109, Australia
{Rolf.Schwitter|Stephen.Guy}@mq.edu.au

Abstract. Answer Set Programming is a compelling non-monotonic knowledge representation paradigm for representing specifications in controlled natural language and reasoning about them. In this paper, we introduce the controlled natural language PENG^{ASP} and discuss the kind of answer set programs that the PENG^{ASP} system automatically generates for a given specification. The controlled natural language PENG^{ASP} is unique, since it is the first controlled natural language that uses Answer Set Programming as target language for reasoning, in particular for question answering. PENG^{ASP} allows us to specify factual and terminological knowledge, to combine weak and strong negation in order to specify a local form of the closed world assumption, to deal with cardinality constraints, to specify arithmetic operations, and to express defaults and exceptions. An emerging PENG^{ASP} specification can be queried in controlled natural language using closed world and open world reasoning depending on the information available in the specification.

1 Introduction

Controlled natural languages (CNLs) are simplified forms of natural language that are constructed from full natural languages by restricting the size of the grammar as well as the vocabulary in order to reduce or eliminate ambiguity and complexity [14, 21]. CNLs can be used as high-level interface languages to knowledge systems to improve the knowledge acquisition and specification process; in particular, if the writing of a specification in CNL is supported by a sophisticated authoring tool with good runtime feedback mechanisms [10]. Although a lot of progress has recently been made in the domain of data-driven applications, there exists still a strong need for mechanisms that support the manual acquisition and encoding of fine-grained commonsense and domain knowledge so that this knowledge can be used for automated reasoning.

There exist a number of CNLs [3, 5, 24] that have been used for knowledge acquisition and for writing specifications but apart from our controlled language PENG^{ASP} none of these languages uses Answer Set Programming (ASP) as target language for knowledge representation and reasoning. The predecessor of PENG^{ASP}, PENG Light [24], used first-order predicate logic as target language, similar to Attempto Controlled English [5], but was not able to deal with non-monotonic forms of reasoning. ASP is interesting in our context, since it offers a declarative modeling language with modeling constructs that allow us to process non-monotonic theories and answer questions over these theories.

In the wider context of natural language processing, ASP has been studied – among other things – for processing sentences with normatives and exceptions [2], for processing biomedical queries [4], for recognising textual entailment [15], for integrating syntactic parsing with semantic disambiguation [15], for parsing Combinatory Categorial Grammar [19], for reasoning with the output of a semantic parser [22], and for domain-specific question answering [23].

In this paper, we will focus on those ASP programs that the language processor of the PENG^{ASP} system generates for a specification written in controlled natural language and show how these programs can be used for question answering. The rest of this paper is structured as follows: In Section 2, we present a brief introduction to ASP, followed by an overview of the controlled natural language PENG^{ASP} in Section 3. In Section 4, we introduce the machinery that is used to process a PENG^{ASP} specification. In Section 5, we present an example specification written in PENG^{ASP}, show how factual and terminological knowledge can be expressed in this controlled language, and discuss how we can specify a local form of the closed world assumption on the level of the controlled language. In Section 6, we have a closer look at how a specification can be investigated with the help of questions and how these questions are represented in ASP. In Section 7, we discuss additional features of the PENG^{ASP} language and show how ordinal and cardinal numbers can be used, how arithmetic operations can be specified, and finally how defaults and exceptions can be expressed. Finally, in Section 8, we conclude and highlight the advantages of our approach.

2 Answer Set Programming

ASP is a declarative knowledge representation language that has its roots in logic programming and non-monotonic reasoning [1, 6, 17]. In ASP, a problem specification is expressed in the form of an extended logic program [8] so that its logical models (= answer sets) provide a solution to the original problem. While ASP programs look at first glance similar to Prolog programs, they use a completely different computational mechanism. In ASP, solutions are computed in a bottom-up fashion and represented as answer sets. An ASP program consists of a finite set of rules of the following form:

$$\mathtt{L}_0 \ \mathtt{;} \ \ldots \ \mathtt{;} \ \mathtt{L}_k \ \mathtt{:-} \ \mathtt{L}_{k+1}, \ \ldots, \ \mathtt{L}_m, \ \mathtt{not} \ \mathtt{L}_{m+1}, \ \ldots, \ \mathtt{not} \ \mathtt{L}_n.$$

where all $\mathtt{L}_i$'s are literals. A literal is either a positive atom or a negative atom. The symbol '`:-`' stands for an if connective and the symbol'`,`' for a conjunction. The expression on the left-hand side of the if connective is called the *head* of the rule, and the expression on the right-hand side is called the *body* of the rule. The head may consist of an epistemic disjunction [9] of literals denoted by the symbol '`;`'. Literals in the body may be preceded by default negation (= negation as failure) denoted by the symbol '`not`'.[1] The head or the body of a rule can be empty. A rule with an empty head is called an *integrity constraint* and a rule with an empty body is called a *fact*. ASP is supported by powerful tools; for example, the new *clingo* 4 system [7] extends ASP's input language by an embedded scripting language that we use for the activation of those parts of a domain-independent background theory that are relevant for a given specification.

3 The Controlled Natural Language PENG^{ASP}

The controlled natural language PENG^{ASP} distinguishes four types of sentences: declarative sentences such as (1), conditional sentences such as (2), imperative sentences such as (3), and questions such as (4):

1. *Pete holds a casual job as tutor.*
2. *If a student is not provably employed then the student is not employed.*
3. *Exclude that Sue teaches Web Technology.*
4. *Are most students employed?*

The syntactic structure of these sentences is derived from a sentence pattern for simple declarative sentences that consists of a subject, a verb, depending complements and optional adjuncts. Each sentences has at least a subject and a verb. Complements depend on the verb and are necessary to complete the meaning of the verb. Adjuncts modify the verb and are optional. All other sentence types are derived from this pattern through coordination, subordination, quantification, weak and strong negation, keywords, and in the case of certain constructions through syntactic movement of elements. The syntactic form of these sentences and the anaphoric use of nominal expressions is carefully restricted by a unification-based grammar that consists of about 350 grammar rules and a domain specific lexicon. The syntactic restrictions of PENG^{ASP} are justified by the formal properties of the ASP language. It is important to note that the writing of a specification in PENG^{ASP} is supported by a sophisticated authoring tool [11] that displays lookahead information which enforces the syntactic structure of the language, and thus makes sure that every PENG^{ASP} sentence is syntactically correct.

4 Processing a PENG^{ASP} Specification

The language processor of the PENG^{ASP} system uses a chart parser and translates a textual specification during the parsing process into a discourse representation structure (DRS) in the spirit of Kamp and Reyle [13]. In contrast to Kamp and Reyle, our extended discourse representation structures (eDRSs) allow for two additional operators: one for constraints and one for weak negation

[1] Some ASP tools [7] allow for double default negated literals in the body of rules, but this construction is not used in the context of PENG^{ASP}.

(= default negation). During the construction of an eDRS, the language processor resolves anaphoric expressions, produces a paraphrase for the input text that clarifies the interpretation of the machine, and generates lookahead information for the authoring tool similar to PENG Light [24]. The generated eDRS basically serves as an interlingua between the controlled natural language and the ASP program. The language processor analyses the eDRS and translates it into an executable ASP program. During this translation process the relevant parts of a predefined domain-independent ASP background theory are identified and a Lua script [12] that registers these parts is dynamically generated and embedded into the resulting ASP program. This script will activate the identified parts of the background theory when the ASP tool *clingo* [7] executes the ASP program.

5 A PENG^{ASP} Specification in a Nutshell

In the following, we will develop a simple specification in PENG^{ASP} in order to highlight some benefits of ASP for CNL processing.

5.1 Specifying Factual Knowledge

The text below is written in PENG^{ASP} and consists of four declarative sentences that convey factual knowledge:

5. *Pete holds a casual job as tutor. Lin is enrolled in a Mathematics degree. Dave is enrolled in an English degree and holds a casual job as guard. Mary who is matriculated in a Computer Science degree holds a casual job as shop assistant.*

The first two sentences are simple declarative sentences, the third sentence contains a coordinated verb phrase, and the fourth sentence an embedded relative clause. The translation of this specification via an eDRS results in an ASP program consisting of a number of facts (only the translation of the third sentence is displayed here for reasons of space):

```
#program base.

const(dave).
inst(sk2, english_degree).
prop(dave, sk2, enrolled_in).
inst(guard, casual_job).
const(guard).
pred(dave, guard, hold).
```

Note that we use a flat representation for logical atoms in ASP and a small number of predefined predicates (such as `const`, `inst`, `prop`, `pred`, etc.). Names are represented as constants and anonymous names as Skolem constants (for example, `sk2`). Skolem constants denote existentially quantified objects and are created during the translation process. Note also that an ASP program can be organised in multiple program parts in the ASP tool *clingo*. The directive `#program base.` indicates the main part of the program.

5.2 Specifying Terminological Knowledge

We can use the controlled language PENG^{ASP} to add terminological knowledge to our specification, for example:

6. *Dave, Mary, Pete and Lin are students.*
7. *Every Computer Science degree, every English degree and every Mathematics degree is a degree program.*
8. *Every casual job is a job.*
9. *Every student who is matriculated in a degree program is enrolled in that degree program.*
10. *If a student holds a job then the student is employed.*

Sentence (6) expresses a series of concept assertions; sentence (7) and sentence (8) state general inclusion axioms; sentence (9) specifies a subsumption relationship between two properties, incl. domain and range restrictions; and sentence (10) relates a binary relation to a property. For example, the translation of sentence (8) adds a rule and a fact of the following form to the ASP program:

```
inst(A, job) :- inst(A, casual_job).

is_subclass(casual_job, job).
```

The rule determines the instances of the class hierarchy and the fact is used to generate the actual class hierarchy with the help of the following part of the domain-independent background theory:

```
#program is_subclass.

subclass(Class1, Class2) :- is_subclass(Class1, Class2).

subclass(Class1, Class3) :- is_subclass(Class1, Class2),
  subclass(Class2, Class3).

-is_leaf(Class2) :- subclass(Class1, Class2).

is_defined(Inst) :- inst(Inst, Class), not -is_leaf(Class).
```

With the help of this background knowledge, the generated answer set will tell us which classes are not leaves and which instances are defined. This information will be used – as we will discuss in Section 6 – for the question answering process.

5.3 Specifying the Local Closed World Assumption

Combining weak and strong negation in a conditional sentence allows us to express a refined form of the closed world assumption in PENG^{ASP}. This assumption states that a predicate does not hold, whenever it cannot be shown (proven) that it does hold [9, 18], for example:

11. *If a student is not provably employed then the student is not employed.*

This local form of the closed world assumption specifies that we have complete information about the predicate *employed* in our knowledge base. In theory, this sentence can be translated into the following ASP rule with a weak negation in the body of the rule and a strong negation in the head:

```
-prop(A, employed) :- inst(A, student),  not prop(A, employed).
```

However, in PENG^{ASP} we split up this kind of rule into two rules during the translation process in order to record for which predicates the local closed world assumption (`local_cwa/1`) has been applied to, in our case:

```
-prop(A, employed) :- local_cwa(neg_prop(A, employed)).

local_cwa(neg_prop(A, employed)) :- inst(A, student),
  not prop(A, employed).
```

As we will see in the next section, this will help us to answer questions under the local closed world assumption as well as under the open world assumption depending on the information that is available in the specification. Note that an ASP program can specify the local closed world assumption for some of its predicates and leave the other predicates for which we do not have complete information in the scope of the open world assumption.

6 Investigating a Specification

The PENG^{ASP} system allows us to investigate an emerging specification with the help of *yes/no*-questions such as (12) and (13) and *wh*-questions such as (14) and (15):

12. *Is Dave who is enrolled in an English degree employed?*
13. *Are most students employed?*
14. *Which students are employed?*
15. *Who is employed?*

In the simplest case, one can translated a question such as (14) into a rule with a specific answer literal (`ans/1`) in the rule head and add this rule to the ASP program:

```
ans(A) :- inst(A, student), prop(A, employed).
```

After the model generation process, all answer literals can be easily extracted from the answer set(s). However, this solution does not tell us whether a question has been answered under closed world reasoning or open world reasoning, and we can not make this distinction clear on the level of the controlled natural language. Since all those predicates for which we have complete information are recorded (as explained in Section 5.3), we can use this information in the question answering process. For example, the positive *wh*-question (15) is automatically translated into a fact and four rules in PENG^{ASP}:

```
ans_id(no1, who, pos).

ans(no1, inst(A, B), pos, cwa) :-  query(A, who, B),
 prop(A, employed), local_cwa(neg_prop(_, employed)).

ans(no1, inst(A, B), pos, owa) :- query(A, who, B),
  prop(A, employed), not local_cwa(neg_prop(_, employed)).

ans(no1, inst(A, B), neg, cwa) :- query(A, who, B),
  -prop(A, employed), local_cwa(neg_prop(A, employed)).

ans(no1, inst(A, B), neg, owa) :- query(A, who, B),
  -prop(A, employed), not local_cwa(neg_prop(A, employed)).
```

The fact (`ans_id/3`) serves as a rule identifier. The first two rules distinguish between positive questions (`pos`) under the local closed world assumption (`cwa`) and the open world assumption (`owa`). The second two rules can be used for a form of cooperative question answering that is triggered if only negative information (`neg`) is available for a positive question. During the translation of the question into ASP, the type of the query word (`who`) is identified and used to activate the relevant part of the background theory. In our example, this part consists of the following rules for question (15):

```
#program who.

ans(ID, unknown, Pol, nil) :- ans_id(ID, who, Pol),
  not ans(ID, _, Pol, cwa), not ans(ID, _, Pol, owa).

query(Inst, who, Class) :- inst(Inst, Class), not -is_leaf(Class).
query(Inst, who, nil)   :- const(Inst), not is_defined(Inst).
```

The first rule deals with the case where no information under closed world and open world reasoning is available to answer the question. The next two rules are used to extract for a given query word instances of classes that occur as leaves of the class hierarchy and named instances that have not been defined in the class hierarchy.

Answering certain types of questions requires counting. For example, question (13) is translated into a fact and a number of rules with a specific literal (`eval/4`) in the head that will trigger the subsequent evaluation process by background axioms:

```
ans_id(no2, most, pos)

eval(no2, A, pos, cwa) :-  inst(A, student), prop(A, employed),
  local_cwa(neg_prop(_, employed)).

eval(no2, A, neg, cwa) :- inst(A, student), -prop(A, employed),
  local_cwa(neg_prop(A, employed)).
```

The first rule takes care of instances that occur in a positive relation and the second rule of instances that occur in a negated relation (here under the closed world assumption). These instances are counted and compared with the help of additional background axioms (for reasons of space, we only show the background axiom for a `yes`-answer):

```
#program most.

ans(ID, yes, pos, cwa) :-
  ans_id(ID, most, pos), C1 > C2,
  C1 = #count { Inst1 : eval(ID, Inst1, pos, cwa) },
  C2 = #count { Inst2 : eval(ID, Inst2, neg, cwa) }.
```

The translation of a specification into an ASP program can result in one or more answer sets. If we end up with more than one answer set, then we can answer a query under brave reasoning or under cautious reasoning. A query is bravely true for a substitution of variables, if its conjunction of body literals is satisfied in at least one answer set, and cautiously true, if it is satisfied in all answer sets.

7 Additional Features of PENG^{ASP}

In the following subsections, we discuss additional features of the language PENG^{ASP} that are novel in controlled natural language processing and can be conveniently represented and processed in ASP.

7.1 Ordinal and Cardinal Numbers

PENG^{ASP} allows us to express statements that contain ordinal and cardinal numbers and to anaphorically link noun phrases with ordinal numbers (e.g., *first course*) to noun phrases that introduce a cardinality restriction (e.g., *two courses*):

16. *There are exactly two courses. The first course is Web Technology and the second course is Information Technology. Alice and Sue are lecturers. Every lecturer teaches exactly one course.*

This partial specification (16) generates four different answer sets, since we neither specified that the two courses must be distinct nor who teaches which course:

```
Answer: 1
pred(alice, information_technology, teach)
pred(sue, information_technology, teach) ...
Answer: 2
pred(alice, web_technology, teach)
pred(sue, web_technology, teach) ...
```

```
Answer: 3
pred(alice, information_technology, teach)
pred(sue, web_technology, teach) ...
Answer: 4
pred(alice, web_technology, teach)
pred(sue, information_technology, teach) ...
```

We can exclude the first two answer sets with the help of the following statement in PENG^{ASP}:

17. *Every lecturer teaches exactly one distinct course.*

This sentence contains the predefined keyword *distinct* in object position and is a short form for the subsequent two sentences:

18. *Every lecturer teaches exactly one course.*
19. *For every course there is exactly one lecturer who teaches that course.*

The translation of sentence (17) as well as the translation of the alternatives (18) and (19) results in the following two ASP rules with cardinality constraints in the rule head:

```
1 { pred(A, B, teach) : inst(A, lecturer) } 1 :- inst(B, course).

1 { pred(B, A, teach) : inst(A, course) } 1 :- inst(B, lecturer).
```

This information excludes the first and the second answer set where both lecturers teach the same course; we can add further information to our specification to exclude one of the two remaining answer sets, for example:

20. *Alice teaches Web Technology.*

Now, we end up with one answer set where Alice teaches Web Technology and Sue teaches Information Technology. We can achieve the same effect by replacing (20) by an imperative sentence such as:

21. *Exclude that Sue teaches Web Technology.*

This imperative sentence translates into a constraint of the form:

```
:- pred(sue, web_technology, teach).
```

7.2 Arithmetic Operations

PENG^{ASP} allows us to specify arithmetic operations in the antecedent of conditional sentences and in imperative sentences (constraints). Numeric variables (e.g., *N1* and *N2*) that occur in these arithmetic expressions can be used anaphorically and can take part in arithmetic operations:

22. *Sue sits in the first office and Alice sits in the second office.*
23. *If there is an office N1 and there is an office N2 and N2 is equal to N1 plus 1 then the office N2 is right of the office N1.*

Sentence (23) also illustrates how an arithmetic operation can play a defining role in the description of a particular expression (*right of*).

7.3 Defaults and Exceptions

Default assumptions are necessary in situations where the information is incomplete but where we still need to be able to draw tentative conclusions [18]. We may be forced to withdraw these conclusions later when new information becomes available. Below is an example specification that states a default and two exceptions to this default in PENG^{ASP}; this example is similar to the ASP example in [9] but here reconstructed in controlled language:

24. *Students are normally afraid of math.*
25. *If a student is enrolled in a Mathematics degree then the student is not afraid of math.*
26. *If a student is not provably not enrolled in a Computer Science degree then the student is abnormally afraid of math.*
27. *If there is a degree program X1 and there is a degree program X2 and a student is enrolled in the degree program X1 and X1 is not the same as X2 then the student is not enrolled in the degree program X2.*

The first sentence (24) specifies a default, the second sentence (25) a strong exception to that default, the third sentence (26) a weak exception, and finally sentence (27) negative information for degree programs and students using the local closed world assumption (with the help of two string variables: *X1* and *X2*). This specification is automatically translated by the PENG^{ASP} system into the following ASP program:

```
const(math).

prop(A, math, afraid_of) :-
  inst(A, student),
  not ab(d_afraid_of(A, math)),
  not -prop(A, math, afraid_of).

-prop(B, math, afraid_of) :-
  inst(B, student),
  inst(C, mathematics_degree),
  prop(B, C, enrolled_in).

ab(d_afraid_of(D, math)) :-
  inst(D, student),
  inst(E, computer_science_degree),
  not -prop(D, E, enrolled_in).

-prop(F, G, enrolled_in) :-
  inst(H, degree_program),
  inst(G, degree_program),
  inst(F, student),
  prop(F, H, enrolled_in),
  F != G.
```

Note two things here: First, the keyword *normally* in (24) triggers a default rule that applies if it is not provable that a students is abnormally afraid of math (week exception) and if it is not provable that a student is not afraid of math (strong exception). Second, the expression *not provably not enrolled in* in (26) is translated into a weak negation followed by a strongly negated literal in ASP.

8 Conclusion

PENG^{ASP} is a fully implemented system that translates specifications written in controlled natural language via extended discourse representation structures into executable ASP programs. The controlled natural language PENG^{ASP} serves as a high-level specification language to ASP programs and can be used to investigate an emerging specification. We argued that ASP is an attractive non-monotonic knowledge representation language for controlled language processing that allows us in contrast to other existing controlled languages to write non-monotonic specifications. PENG^{ASP} allows us to combine weak and strong negation to specify a local form of the closed world assumption, to express integrity and cardinality constraints, to define background theories, to execute arithmetic operations, and to express defaults and exceptions. ASP generates flat models that can be directly used for the question answering process. The PENG^{ASP} systems translates questions into rules with specific answer literals in the head. The question answering process is supported by domain-independent background axioms that are selectively activated during the translation of a discourse representation structure into an ASP program.

References

1. Baral, C.: *Knowledge Representation, Reasoning and Declarative Problem Solving,* Cambridge University Press (2003)
2. Baral, C., Dzifcak, J., Son, T.C.: Using Answer Set Programming and Lambda Calculus to Characterize Natural Language Sentences with Normatives and Exceptions. In: *Proceedings of the Twenty-Third Conference on Artificial Intelligence (AAAI 2008)*, pp. 818-823 (2008)
3. Clark, P., Harrison, P., Jenkins, T., Thompson, J., Wojcik, R.: Acquiring and Using World Knowledge using a Restricted Subset of English. In: *The 18th International FLAIRS Conference (FLAIRS'05)*, pp. 506-511 (2005)
4. Erdem, E., Yeniterzi, R.: Transforming Controlled Natural Language Biomedical Queries into Answer Set Programs. In: *Proceedings of the Workshop on BioNLP*, pp. 117-124, (2009)
5. Fuchs, N.E., Kaljurand, K., Kuhn, T.: Attempto Controlled English for Knowledge Representation. In: C. Baroglio, P. A. Bonatti, J. Maluszynski, M. Marchiori, A. Polleres, S. Schaffert, (eds.), *Reasoning Web*, Fourth International Summer School 2008, LNCS 5224, pp. 104-124 (2008)
6. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: *Answer Set Solving in Practice.* In: Synthesis Lectures on Artificial Intelligence and Machine Learning, Vol. 6, No. 3, pp. 1-238 (2012)

7. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: *Clingo = ASP + Control: Extended Report*. To appear in Theory and Practice of Logic Programming (2015)
8. Gelfond, M., Lifschitz, V.: The Stable Model Semantics for Logic Programming. In: *Proceedings of the Fifth International Conference on Logic Programming (ICLP)*, pp. 1070-1080 (1988)
9. Gelfond, M., Yulia Kahl, Y.: *Knowledge Representation, Reasoning, and the Design of Intelligent Agents, The Answer-Set Programming Approach*, Cambridge University Press (2014)
10. Gunning, D., Chaudhri, V.K., Clark, P., Barker, K. et al.: Project Halo Update - Progress Toward Digital Aristotle. In: *AI Magazine*, Vol. 31, No. 3 (2010)
11. Guy, S., Schwitter, R.: Architecture of a Web-based Predictive Editor for Controlled Natural Language Processing. In: B. Davis et al. (eds.): *CNL 2014*, LNAI 8625, pp. 167-178 (2014)
12. Ierusalimschy, R.: Programming in Lua. Lua.org; 3 Edition (2013)
13. Kamp, H., van Genabith, J., Reyle, U.: Discourse Representation Theory. In: D. Gabbay and F. Guenthner (eds.), *Handbook of Philosophical Logic*, Vol. 15, pp. 125-394 (2011)
14. Kuhn, T.: A Survey and Classification of Controlled Natural Languages. In: *Computational Linguistics*, Vol. 40, No. 1, pp. 121-170 (2014)
15. Lierler, Y., Lifschitz, V.: Logic Programs vs. First-Order Formulas in Textual Inference. In: *Proceedings of the 10th International Conference on Computational Semantics (IWCS)* (2013)
16. Lierler, Y., Schüller, P.: AspCcgTk: Towards Syntactic Parsing with Semantic Disambiguation by Means of Declarative Programming. In: *Association for Logic Programming (ALP), Newsletter*, December 31 (2014)
17. Lifschitz, V.: What is Answer Set Programming? In: *Proceedings of AAAI 2008*, pp. 1594-1597 (2008)
18. Reiter, R.: A Logic for Default Reasoning. In: *Artificial Intelligence*, Vol. 13, pp. 81-132 (1980)
19. Schüller, P.: Flexible Combinatory Categorial Grammar parsing using the CYK algorithm and Answer Set Programming. In: Pedro Cabalar and Tran Cao Son (ed), *Logic Programming and Nonmonotonic Reasoning (LPNMR)*, LNCS 8148, pp. 499-511 (2013)
20. Schwitter, R.: Controlled Natural Languages for Knowledge Representation. In: *Proceedings of COLING 2010*, Beijing, China, pp. 1113-1121 (2010)
21. Schwitter, R.: Controlled Natural Language. In: Sin-Wai Chan (ed.), *Routledge Encyclopedia of Translation Technology*, Chapter 28, Routledge, Taylor & Francis Group (2015)
22. Sharma, A., Vo, N.H., Somak, A., Baral, C.: Towards Addressing the Winograd Schema Challenge – Building and Using a Semantic Parser and a Knowledge Hunting Module. In: *Proceedings of IJCAI 2015*, Buenos Aires, Argentina, pp. 1319-1325, (2015)
23. Todorova, Y., Gelfond, M.: Toward Question Answering in Travel Domains. In: E. Erdem et al. (ed.), *Correct Reasoning*, LNCS 7265, pp. 311-326 (2012)
24. White C., Schwitter, R.: An Update on PENG Light. In: L. Pizzato and R. Schwitter (eds.), *Proceedings of ALTA 2009*, Sydney, Australia, pp. 80-88 (2009)

Default Reasoning with Propositional Encoding of Topological Relations

Przemysław Andrzej Wałęga

University of Warsaw, Institute of Philosophy, Poland
`przemek.walega@wp.pl`

Abstract. We present a formal system that enables to perform an effective default reasoning with topological relations. Our approach extends the well-known propositional Default Logic with a possibility to specify two sets of information, namely positive (that has to be true) and negative (that cannot be true). This distinction provides a significant increase of expressive power of set-theoretical interpretation of propositional formulae. Additionally to a standard fixed-point semantics of an extension we provide its operational semantics and prove equivalence of the abovementioned. We use the latter to establish an effective reasoning algorithm and implemented in Prolog. As a proof of concept we demonstrate the system's application to geographic information system with real data.

Keywords: Default Reasoning, Qualitative Spatial Reasoning, Commonsense Reasoning

1 Introduction

Formal representation and reasoning about space is recognized as a crucial part of commonsense reasoning and knowledge representation. Recently, a strong need for non-monotonic and default reasoning in spatial domain has been recognized [3] and a number of such methods established. Shanahan [17] described default reasoning for space occupancy problem and aformalized a default rule "space is normally empty" for systems dealing with incomplete spatial knowledge. Möller and Wessel [12] presented terminological default rules obtained by means of description logic terms rather than first-order formulae. Hartley [10] and Hazarika [11] worked on continuous aspects of space and more recently, methods for abductive spatial reasoning have been presented, e.g., in [7]. The main fields of applications of non-monotonic and default spatial reasoning methods are geographic information systems (GIS), computer-aided architecture design (CAAD) systems, cognitive spatial systems, visual interpretation and cognitive robotics. In the flagship applications field, i.e., GIS, modern systems possess powerful quantitative tools and are able to perform complex numerical transformation but have very limited qualitative capabilities. However, since GIS often deal with imprecise and incomplete knowledge while performing geospatial reasoning about distance, direction, topology and shape, qualitative methods such as the qualitative spatial reasoning (QSR) [4] are highly recommended. The abovementioned need is well-known among researchers and resulted in establishment of a number of QSR methods especially for GIS systems [20,12,6]. In this paper we present a new formal method that enables to perform effective default reasoning with (qualitative) topological information (e.g., Region Connection Calculus described below) that may be applied, e.g., to geospatial reasoning systems.

▷ **Region Connection Calculus.** One of the main topological QSR methods is Region Connection Calculus ($\mathcal{RCC}$) [14]. The $\mathcal{RCC}$ theory is based on a binary, reflexive and symmetric relation $C(A, B)$ defined over non-null regions, that is interpreted as "A is connected with B". The relation is used to define base relations, i.e., jointly exclusive and pairwise disjoint (JEPD) binary topological relations between regions. The $\mathcal{RCC}$-5 introduces 5 base relations, namely "A is discrete with B", "A partially overlaps B","A is a proper part of B", "B is a proper part of A" and "A is equal to B" as depicted in Fig. 1. $\mathcal{RCC}$ provides a simple look-up mechanism for composition of the base relations and is often used for topological reasoning with incomplete information.

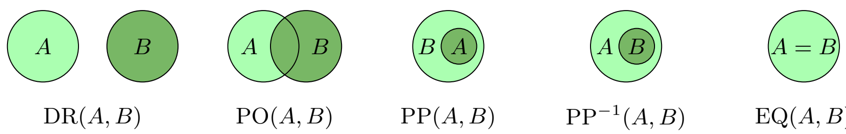

Fig. 1: Base relations of $\mathcal{RCC}$-5.

▷ **Default Reasoning.** Default reasoning constitutes a crucial feature of nonmonotonic systems. One of the best known default reasoning formalisms was proposed by Reiter in [15] and is known as Default Logic. It augments classical first-order logic by default rules of the form $\frac{\alpha:\beta_1,\ldots,\beta_n}{\gamma}$, where $\alpha, \beta_1, \ldots, \beta_n, \gamma$ are first-order formulae and $n \in \mathbb{N}$. The intuitive meaning of the rule is "if α is true and $\beta_1, \ldots, \beta_n$ are consistent with the knowledge then by default conclude γ". A set of facts obtained from the initial knowledge and proper applying of defaults is called an extension. Default Logic is deeply studied [1,8] and a number of its variants and modifications have been proposed [5], e.g., Statistical Default Logic [19], General Default Logic [21] or Distributed Default Logic [16].

In this paper we present a new formal default reasoning system based on propositional logic and capable to perform topological reasoning. The use of propositional logic makes the reasoning relatively simple but on the other hand, it is expressive enough to represent interesting topological relations, e.g., all $\mathcal{RCC}$-5 relations. The paper is organized as follows. In Section 2 we describe topological interpretation of propositional logic. In Section 3 we present modified definitions of crucial notions, namely a default rule, a default theory and an extension of a default theory. We provide fixed-point and operational semantics of an extensions and show their equivalence. In Section 4 we present a method for automated reasoning and in Section 5 a proof of the concept case-study for GIS application. Finally, in Section 6 we conclude the paper and indicate our future work.

2 Topological Interpretation of Propositional Calculus

It is well-known [13,18] that there is a connection between formulae of propositional logic and set-terms. Let the propositional letters denote arbitrary subsets of a non-empty set V (universe), whereas the connectives $\neg, \wedge, \vee$ denote set operations of complement, intersection and union respectively. Formally, the set-theoretical interpretation of propositional logic is obtained by means of a model $\langle V, \Sigma, d \rangle$, where V is an arbitrary non-empty set, $\Sigma = \{A, B, C, \dots\}$ is a denumerably infinite set of propositional constants and $d : \Sigma \to \mathcal{P}(V)$ assigns to each propositional constant a subset of V. Additionally, d is extended to all propositional well formed formulae (wffs) as follows:

$$d(\neg\phi) = \overline{d(\phi)}\ , \quad d(\phi \wedge \psi) = d(\phi) \cap d(\psi)\ , \quad d(\phi \vee \psi) = d(\phi) \cup d(\psi)\ ,$$

where ϕ, ψ are propositional wffs, e.g., $d(\neg A \wedge B \to C) = \overline{\overline{d(A)} \cap d(B)} \cup d(C)$. Then, a propositional formula ϕ is a tautology iff in all models $d(\phi) = V$. It is shown in [2] that for any propositional wffs $\phi_1, \dots, \phi_n, \psi$ the following holds $\phi_1, \dots, \phi_n \models \psi$ iff in any model $\langle V, \Sigma, d \rangle$ such that $d(\phi_1) = V, \dots, d(\phi_n) = V$ hold, $d(\psi) = V$ also holds. As a result we are able to perform set-theoretical reasoning by means of propositional calculus.

2.1 Propositional Encodings of Spatial Relations

In the abovementioned method it cannot be expressed that some set is not equal to V. Hence, it cannot be stated that, e.g., set $d(A)$ is not empty or that $d(A)$ is not a subset of $d(B)$. In order to increase the expressive power we adopt a method presented in [2], where an additional set of propositional formulae is involved and interpreted as a set of set-terms that are not equal to V. As an example, a pair of propositional formulae sets $(\{\neg(A \wedge B)\}, \{\neg A, \neg B\})$ corresponds to the following pair $(\{\overline{d(A) \cap d(B)} = V\}, \{\overline{A} \neq V, \overline{B} \neq V\})$ and has an intuitive meaning that sets $d(A)$, $d(B)$ are non-empty and discrete. We call the former set of propositional formulae a set of model constraints, whereas the latter a set of entailment constraints.

We interpret V as a 2-dimensional space, and set-terms as regions of V. Then, the presented method enables us to interpret model constraints together with entailment constraints as topological relations between spatial regions. It turns out that such an interpretation enables to model, e.g., all atomic relations of $\mathcal{RCC}$-5, as presented in Table 1. In what follows, to shorten the notation we use the same notation for a propositional letter and a corresponding spatial region, e.g., a region corresponding to a propositional constant A will be denoted also by A. It should be fairly clear from the context when we mean a propositional letter and when a region.

2.2 Spatially Possible Configurations

We call pair consisting of a set of model constraints and a set of entailment constraints a spatial configuration (configuration in short). Some configurations are

Table 1: Representation of base $\mathcal{RCC}$-5 relations.

Relation	Description	Model Constraints	Entailment Constraints
DR(A, B)	A and B are discrete	$\neg(A \wedge B)$	$\neg A, \neg B$
PO(A, B)	A partially overlaps B		$\neg A \vee \neg B, A \rightarrow B,$ $A \rightarrow B, \neg A, \neg B$
PP(A, B)	A is a proper part of B	$A \rightarrow B$	$B \rightarrow A, \neg A, \neg B$
PP$^{-1}(A, B)$	B is a proper part of A	$B \rightarrow A$	$A \rightarrow B, \neg A, \neg B$
EQ(A, B)	A is equal to B	$A \equiv B$	$\neg A, \neg B$

spatially possible, whereas others are not. More formally, we call a configuration impossible iff there is no assignment of sets to propositional letters occurring in the configuration such that all set-terms corresponding to the configuration hold. Otherwise, we call the configuration possible. In the following sections we use a property proved in [2] that a configuration is impossible iff there is some propositional formula ϕ occurring in entailment constraints, such that model constraints propositionally entail ϕ. This result enables us to obtain an effective method (reasoning in propositional logic) for checking if a configuration is possible.

3 Propositional Default Logic with Topological Relations

In this section we introduce a default reasoning method with two sets of propositional formulae. At first we present preliminary definitions and then fixed-point and operational semantics of an extension.

3.1 Preliminary Definitions

▷ **Language.** Let $\mathcal{A}$ be an alphabet consisting of countably many propositional constants $A, B, C, \dots$ logical constants $\neg, \wedge, \vee, \rightarrow, \equiv$ and punctuation signs. We work with a propositional language L that consists of wffs over the alphabet $\mathcal{A}$ (denoted by $\phi, \psi, \dots$). As usual, for any set of wffs S and any wffs ϕ, by $S \vdash \phi$ we denote that ϕ is propositionally provable from S. We define for any set of wffs S, theory of S as $Th_L(S) = \{\phi \mid \phi \in L \text{ and } S \vdash \phi\}$.
▷ **Default Rules.** We introduce a default rule δ as an expression of the form (1), where Φ, Ψ, χ are pairs of sets of propositional wffs. The elements of the pairs are indexed with $+$ and $-$ respectively as presented in (2).

$$\frac{\Phi : \Psi}{\chi} \qquad (1) \qquad\qquad \frac{\Phi : \Psi}{\chi} = \frac{\langle \Phi^+, \Phi^- \rangle : \langle \Psi^+, \Psi^- \rangle}{\langle \chi^+, \chi^- \rangle} \qquad (2)$$

The sets indexed with $+$ are interpreted as sets of model constraints, whereas those indexed with $-$ as sets of entailment constraints. We call Φ a prerequisite, Ψ a justification and χ a conclusion of δ. We denote Φ, Ψ and χ respectively by $pre(\delta)$, $just(\delta)$ and $cons(\delta)$. Additionally, $pre^+(\delta) = \Phi^+$, $pre^-(\delta) = \Phi^-$, $just^+(\delta) = \Psi^+$, $just^-(\delta) = \Psi^-$, $cons^+(\delta) = \chi^+$ and $cons^-(\delta) = \chi^-$. In what follows, to shorten notation, we also denote defaults of the form (1) by $(\Phi : \Psi/\chi)$.

Our definition of a default differs from a classical definition from Default Logic [15] however, in this paper by a default (or a default rule) we always mean a rule of the form (1).

▷ **Default Theory.** A default theory in our approach is a tuple $T = \langle W^+, W^-, D\rangle$ such that W^+, W^- are (finite or infinite) sets of propositional wffs and D is a (finite or infinite) set of defaults. We interpret W^+ and W^- as initial sets of model entailment constraints respectively.

3.2 Fixed-point Semantics of Extension

By an extension of the spatial default theory T we intuitively mean a pair consisting of a set of positive beliefs and a set of negative beliefs that are acceptable in T. In what follows we give a formal definition based on a fixed-point.

Definition 1 (extension: fixed-point). *Let $T = \langle W^+, W^-, D\rangle$ be a default theory. For any pair of wffs sets $U = \langle U^+, U^-\rangle$ let $\Gamma(U) = \langle \Gamma^+(U^+), \Gamma^-(U^-)\rangle$ be such that $\Gamma^+(U^+)$, $\Gamma^-(U^-)$ are the smallest sets satisfying the following conditions:*

1. $W^+ \subseteq \Gamma^+(U^+)$,
2. $W^- \subseteq \Gamma^-(U^-)$,
3. $Th_L\Gamma^+(U^+) = \Gamma^+(U^+)$,
4. $Th_L\Gamma^-(U^-) = \Gamma^-(U^-)$,
5. *If* $(\Phi : \Psi/\mathrm{X}) \in D$ *and* $\Phi^+ \subseteq \Gamma^+(U^+)$ *and* $\Phi^- \subseteq \Gamma^-(U^-)$ *and there is no* $\psi \in U^- \cup \Psi^-$ *such that* $U^+ \cup \Psi^+ \vdash \psi$, *then* $\mathrm{X}^+ \subseteq \Gamma^+(U^+)$ *and* $\mathrm{X}^- \subseteq \Gamma^-(U^-)$.

Then, E is an extension for T iff E is a fixed point of Γ, i.e., $\Gamma(E) = E$.

The conditions *1.* and *2.* provide that the initial knowledge (positive and negative) is preserved in E. Then, *3.* and *4.* make E^+ and E^- closed under logical conclusion, whereas *5.* provides that E is closed under application of defaults, i.e., if there is a default $\delta \in D$ that is applicable to E, then $cons^+(\delta)$ are in E^+ and $cons^-(\delta)$ are in E^-. The notion of applicability of a default is used in further sections, therefore we provide its formal definition.

Definition 2 (applicability). *We say that a default $(\Phi : \Psi/\mathrm{X})$ is applicable to $U = \langle U^+, U^-\rangle$ iff:*

1. *Φ is among current beliefs, i.e., $\Phi^+ \subseteq U^+$ and $\Phi^- \subseteq U^-$,*
2. *and Ψ is consistent with U, i.e., there is no $\psi \in U^- \cup \Psi^-$ such that $U^+ \cup \Psi^+ \vdash \psi$.*

We say that a default theory $T = \langle W^+, W^-, D\rangle$ is consistent iff there is no $\psi \in W^-$ such that $W^+ \vdash \psi$. Hence, in a consistent default theory, knowledge $\langle W^+, W^-\rangle$ is a spatially possible configuration.

Proposition 1. *Each extension of a consistent default theory is a (spatially) possible configuration.*

Proof. Let $T = \langle W^+, W^-, D\rangle$ be a consistent default theory and let $E = \langle E^+, E^-\rangle$ be an extension of T. To show, that there is no $\psi \in E^-$ such that $E^+ \vdash \psi$. The statement follows from the condition *5.* of the Definition 1. ■

3.3 Operational Semantics of Extension

The Definition 1 is not constructive and therefore hard to be implemented. In what follows we present an alternative, operational definition of an extension. It is operational in a sense that it provides an algorithmic procedure for computing extensions of a default theory (for an operational semantics of a classical Default Logic see [1]).

For a default theory $T = \langle W^+, W^-, D\rangle$ let $\Pi = \langle \delta_0, \delta_1, \dots\rangle$ be a (finite or infinite) sequence of defaults from D without multiple occurrences. We interpret Π as a possible order of applying defaults. Then, for any Π we define:

- $In^+(\Pi) = Th_L(W^+ \cup \{cons^+(\delta) \mid \delta \text{ occurs in } \Pi\})$ is a current positive knowledge obtained after applying defaults from Π, i.e., formulas believed to be true,
- $In^-(\Pi) = Th_L(W^- \cup \{cons^-(\delta) \mid \delta \text{ occurs in } \Pi\})$ is a current negative knowledge obtained after applying defaults from Π, i.e., formulas believed to be false,
- $Out^+(\Pi) = \bigcup\{just^+(\delta) \mid \delta \text{ occurs in } \Pi\})$ are statements that should no be inconsistent with the positive knowledge even after applying further defaults,
- $Out^-(\Pi) = \bigcup\{just^-(\delta) \mid \delta \text{ occurs in } \Pi\})$ are statements that should no be inconsistent with the negative knowledge even after applying further defaults.

We denote by $\Pi[k]$ the initial segment of Π of length k. Then, we define the notions of a process, a successful and a closed sequence of defaults as follows.

Definition 3. *Let $T = \langle W^+, W^-, D\rangle$ be a default theory and Π a sequence of defaults from D without multiple occurrences. Then,*

- *Π is a process of T iff for each $\delta_k \in \Pi$, $pre^+(\delta_k) \subseteq In^+(\Pi[k])$ and $pre^-(\delta_k) \subseteq In^-(\Pi[k])$, i.e., prerequisites of δ_k are in current knowledge of $\Pi[k]$,*
- *Π is successful iff there is no $\psi \in In^-(\Pi) \cup Out^-(\Pi)$ such that $In^+(\Pi) \cup Out^+(\Pi) \vdash \psi$, otherwise is failed,*
- *Π is closed in T iff there is no default $\delta \in D$ that does not belong to Π and is applicable to $\langle In^+(\Pi), In^-(\Pi)\rangle$.*

Definition 4 (extension: operationally). *We say that E is an extension of a default theory T iff there is a closed and successful process Π of T such that $E = \langle In^+(\Pi), In^-(\Pi)\rangle$.*

▷ **Example 1:** Let $T = \langle W^+, W^-, D\rangle$, $W^+ = \{\neg(A \wedge B)\}$, $W^- = \{\neg A, \neg B, \neg C\}$ and $D = \{\delta_1, \delta_2\}$ such that:

$$\delta_1 = \frac{\langle \emptyset, \emptyset\rangle : \langle\{C \to A\}, \{A \to C\}\rangle}{\langle\{C \to A\}, \{A \to C\}\rangle}, \quad \delta_2 = \frac{\langle \emptyset, \emptyset\rangle : \langle\{C \to B\}, \{B \to C\}\rangle}{\langle\{C \to B\}, \{B \to C\}\rangle}.$$

W^+, W^- have an intuitive meaning that regions A, B, C are non-empty and A is discrete with B. Then, there are 2 closed and successful processes of T, namely $\Pi_1 = \langle \delta_1 \rangle$ and $\Pi_2 = \langle \delta_2 \rangle$. Therefore the extensions are as follows $E_1 = \langle Th_L(W^+ \cup \{C \to A\}), Th_L(W^- \cup \{A \to C\}) \rangle$ and $E_2 = \langle Th_L(W^+ \cup \{C \to B\}), Th_L(W^- \cup \{B \to C\}) \rangle$ as graphically presented in Fig. 2. Let us check that Π_1 is indeed a successful and closed process. We have:

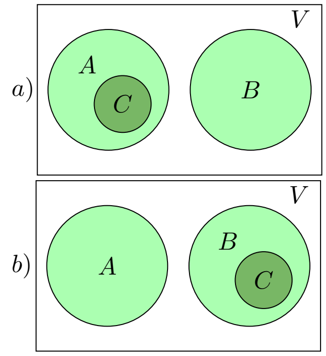

Fig. 2: Extensions of T.

1. $In^+(\Pi_1) = Th(W^+ \cup \{C \to A\})$,
2. $In^-(\Pi_1) = Th(W^- \cup \{A \to C\})$,
3. $Out^+(\Pi_1) = \{C \to A\}$,
4. $Out^-(\Pi_1) = \{A \to C\}$.

Then by the definition Π_1 is indeed a successful and closed process of T. A case for Π_2 is analogous.

The following theorem shows that the Definitions 1 and 4 of an extension (the non constructive and the operational) are equivalent.

Theorem 1. *Let $T = \langle W^+, W^-, D \rangle$ be a default theory. E is an extension of T (in a sense of Definition 4) iff $E = \Gamma(E)$ (for Γ from the Definition 1).*

Proof. First, we prove the implication from left to right. Let Π be a closed and successful process of T, then $E = \langle E^+, E^- \rangle = \langle In^+(\Pi), In^-(\Pi) \rangle$. By the definitions of In^+ and In^- we have $W^+ \subseteq In^+$, $W^- \subseteq In^-$, $Th_L(In^+) = In^+$ and $Th_L(In^-) = In^-$. Since Π is closed, the condition *5*. of the Definition 1 is also fulfilled. Therefore we have $\Gamma^+(E^+) \subseteq E^+$ and $\Gamma^+(E^-) \subseteq E^-$.

Now, we show that $E^+ \subseteq \Gamma^+(E^+)$ and $E^- \subseteq \Gamma^-(E^-)$. We prove inductively that for any $k \in \mathbb{N}$, $In^+(\Pi[k]) \subseteq \Gamma(E^+)$ and $In^-(\Pi[k]) \subseteq \Gamma(E^-)$. For $k = 0$ inclusions are trivial. Suppose that the statement holds for k and let $\delta_{k+1} = (\Phi : \Psi / \chi)$. Since Π is a successful process it follows that δ_{k+1} is applicable to $\Gamma(E)$. Then, $\chi^+ \subseteq \Gamma(E^+)$ and $\chi^- \subseteq \Gamma(E^-)$. Thus, $In^+(\Pi[k+1]) \subseteq \Gamma(E^+)$ and $In^-(\Pi[k+1]) \subseteq \Gamma(E^-)$ which finishes the prove of $\Gamma(E) = E$.

Second, we prove the implication from right to left. Let $E = \Gamma(E)$. To show that there exists a successful and closed process of T. Fix an arbitrary enumeration $\{\delta^0, \delta^1, \dots\}$ of defaults occurring in D. We construct a sequence of defaults Π as follows.

1) If every $\delta \in D$ applicable to $\langle In^+(\Pi[i]), In^-(\Pi[i]) \rangle$ is already in $\Pi[i]$, then finish the construction with $\Pi = \Pi[i]$,

2) else take $\delta^j \in D$ applicable to $\langle In^+(\Pi[i]), In^-(\Pi[i]) \rangle$ with the smallest index j and let $\Pi[i+1] = \Pi[i] \frown \langle \delta^j \rangle$ [1].

[1] As usual, by $\frown$ we denote the concatenation operator, i.e., for all sequences $\overline{a} = \langle a_0, \dots, a_n \rangle$, $\overline{b} = \langle b_0, \dots, b_m \rangle$ the following holds $\overline{a} \frown \overline{b} = \langle a_0, \dots, a_n, b_0, \dots, b_m \rangle$.

Obviously, Π is a closed process of T. We show by induction on i that $\Pi[i]$ is successful (i.e., there is no $\psi \in In^-(\Pi) \cup Out^-(\Pi)$ such that $In^+(\Pi) \cup Out^+(\Pi) \vdash \psi$) and that $In^+(\Pi[i]) \subseteq E^+$ and $In^-(\Pi[i]) \subseteq E^-$. For $i = 0$ both conditions are trivially fulfilled. If the conditions are fulfilled for i and there is $\delta^j \in D$ applicable to $\langle (In^+(\Pi[i]), In^-(\Pi[i]) \rangle$ such that $\delta^j \notin \Pi[i]$ then by the construction step *2*) both conditions are fulfilled for $i + 1$. Hence, Π is a successful and closed process of T and $In^+(\Pi) \subseteq \Gamma(E^+)$ and $In^-(\Pi) \subseteq \Gamma(E^-)$. Now we prove that $\Gamma(E^+) \subseteq In^+(\Pi)$ and $\Gamma(E^-) \subseteq In^-(\Pi)$ by showing that conditions *1. – 5.* of the Definition 1 are fulfilled for Π, i.e., the following hold:

1. $W^+ \subseteq In^+(\Pi)$,
2. $W^- \subseteq In^-(\Pi)$,
3. $Th_L(In^+(\Pi)) = In^+(\Pi)$,
4. $Th_L(In^-(\Pi)) = In^-(\Pi)$,
5. If $(\Phi : \Psi/\chi) \in D$ and $\Phi^+ \subseteq In^+(\Pi)$ and $\Phi^- \subseteq In^-(\Pi)$
   and there is no $\psi \in \Psi$ such that $In^+(\Pi) \cup \Psi^+ \vdash \psi$,
   then $\chi^+ \subseteq In^+(\Pi)$ and $\chi^- \subseteq In^-(\Pi)$.

Conditions 1.-4. follow from the definitions of $In^+(\Pi)$ and $In^-(\Pi)$, whereas condition 5. results from the construction step *1*) of Π. Due to space limitations we omit to show that the construction is proper for infinite processes (which is indeed true and not hard to be shown). ■

4 Automated Reasoning Method

In this section we present an algorithm for computing extensions of a default theory $T = \langle W^+, W^-, D\rangle$. It creates a process tree in which edges are labelled with $\delta \in D$ and vertices are the sequences of so far applied defaults. Hence, the root is an empty sequence and each other vertex v is a sequence Π of edges' labels that create a path from the root to v. The tree is build in such a way, that each vertex is a process of T and new vertices are added until there are no more applicable defaults that have not been applied in a given path. When the tree is built, each leaf is checked if it is an extension. A more precise description of the algorithm is as follows.

1) Create a root of a tree with $\Pi = \langle\ \rangle$.

2) For each leaf Π of a so far constructed tree that is not marked as failed, do what follows. For each $\delta \in D$ applicable to $\langle In^+(\Pi), In^-(\Pi)\rangle$ that has not yet been applied (i.e., $pre^+(\delta) \subseteq In^+(\Pi)$ and $pre^-(\delta) \subseteq In^-(\Pi)$ and there is no $\psi \in \Psi^-$ such that $In^+(\Pi) \cup \Psi^+ \vdash \psi$ and $\delta \notin \Pi$) create a new vertex equal to $\Pi \frown \langle\delta\rangle$ and create an edge labelled with δ from Π to $\Pi \frown \langle\delta\rangle$.

3) For each vertex added in step *2*) check if it is failed (i.e., if there is $\psi \in In^-(\Pi) \cup Out^-(\Pi)$ such that $In^+(\Pi) \cup Out^+(\Pi) \vdash \psi$). If it is the case mark the vertex as failed.

4) Repeat steps *2*) and *3*) until no more vertices are added.

5) Each leaf Π not marked as failed, is marked as successful and closed ($S\&C$ in short) and $\langle In(\Pi), Out(\Pi)\rangle$ is an extension of T.

For each leaf Π marked as $S\&C$ it follows from the construction of the algorithm that Π is a closed and successful process of T, hence $\langle In(\Pi), Out(\Pi)\rangle$ is an extension of T. For finite D the algorithm always terminates, because then there is a finite number of processes (at most $2^{card(D)}$ which is a number of all permutations of D).

▷ **Example 2:** Let $T = \langle W^+, W^-, D\rangle$ be the same as in Example 1. The process tree created by the algorithm is presented in Fig. 3. Each vertex Π of the graph is labelled with $In^+(\Pi)$ and $In^-(\Pi)$ located on the left-hand side and $Out^+(\Pi)$ and $Out^-(\Pi)$ on the right-hand side of the vertex.

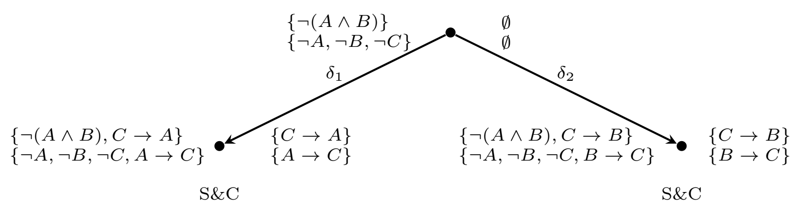

Fig. 3: A process tree for a default theory T from Example 1.

We have implemented the abovementioned algorithm in Prolog. The implementation is straightforward when a procedure determining whether a set of propositional formulae (propositionally) entail another given formula.

5 Application to Geographic Information System

In this section we present a proof of the concept application of our method. We use real data from GIS system, as presented in Fig. 4, in order to reason about possible localization of pubs area in Saxon Garden in Warsaw. The scenario is similar to the dump localization problem proposed in [9] but uses only topological information.

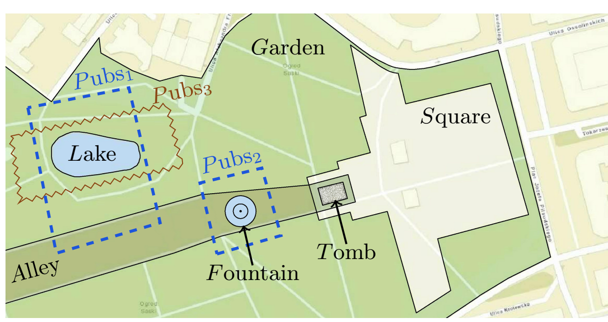

Fig. 4: A Saxon Garden map with possible localizations of a pub area.

The topological representation of the Warsaw Saxon Garden consists of the following regions: the Saxon Garden (G), a lake (L), the Piłsudski Square (S), an alley (A) and a fountain (F) as depicted in Fig. 4. In what follows we use the capital letters to denote the abovementioned regions as well as corresponding propositional constants. The problem to be solved is to decide where should we locate an area with pubs (P). Additionally to the topological information about regions localization (depicted in Fig. 4), we consider a following set of default rules:

- δ_1 - if it is possible then, the pub area should be located around the lake, i.e., $\mathrm{PP}^{-1}(P, L)$, because the lake provides a beautiful view,
- δ_2 - if it is possible then, the pub area should be located around the fountain, i.e., $\mathrm{PP}^{-1}(P, F)$, because the fountain makes the localization more attractive,
- δ_3 - if it is possible then the pub area should be located on the alley, i.e., $\mathrm{PO}(P, A)$, because tourists frequently walk there.

Additionally, the pub area cannot be located around the lake and the fountain at a same time, because it is known that the pub area cannot be so big. Formally, the abovementioned is represented by a default theory $T_1 = (W_1^+, W_1^-, D_1)$, where W_1^+ and W_1^- represent the topological relations between objects, i.e., $W_1^+ = \{L \to G, S \to G, A \to G, F \to A, \neg(L \wedge A), \neg(L \wedge S)\}$, $W_1^- = \{\neg G, \neg L, \neg S, \neg A, \neg F, \neg S \vee \neg A, S \to A, A \to S\}$ and $D_1 = \{\delta_1, \delta_2, \delta_3\}$ correspond to abovementioned default rules, i.e.,:

$$\delta_1 = \frac{\langle \emptyset, \emptyset \rangle : \langle \{L \to P, \neg(F \wedge P)\}, \{P \to L\} \rangle}{\langle \{L \to P, \neg(F \wedge P)\}, \{P \to L\} \rangle},$$

$$\delta_2 = \frac{\langle \emptyset, \emptyset \rangle : \langle \{F \to P, \neg(L \wedge P)\}, \{P \to F\} \rangle}{\langle \{F \to P, \neg(L \wedge P)\}, \{P \to F\} \rangle},$$

$$\delta_3 = \frac{\langle \emptyset, \emptyset \rangle : \langle \emptyset, \{\neg(P \wedge A), P \to A, A \to P\} \rangle}{\langle \emptyset, \{\neg(P \wedge A), P \to A, A \to P\} \rangle}.$$

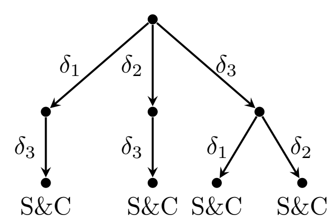

Fig. 5: A process tree for T_1.

Then the algorithm computing extensions creates a process tree presented in Fig. 5. Although there are 4 leaves marked as $S\&C$, there are only 2 different extensions (each extension occurs twice in this process tree). In the first extension L is a proper part of P and A overlaps P, whereas in the second extension F is a proper part of P. The computed localizations of P are denoted by $P\mathrm{ubs}_1$ and $P\mathrm{ubs}_2$ in Fig. 4. The method enables to check if (a) there exists some extension of T_1, (b) a formula ϕ holds in some extension of T_1 (brave reasoning) and (c) a formula ϕ holds in all extensions of T_1 (cautious reasoning). As an example, the method enables to conclude that for some extension L is a proper part of P and for all extensions P partially overlaps A.

Now, let us add the information that there is a Tomb of the Unknown Soldier (T) located inside S and inside A and that the following rule has to be fulfilled:

δ_5 - if the tomb is located on the alley, then the pub area has to be discrete with the alley, because the tomb is a place of worship and should be separated from a place for having fun and parties.

Then, let $T_2 = \langle W_1^+, W_2^-, D_2 \rangle$, where $W_2^- = W_1^- \cup \{\neg T, T \to A, T \to S\}$ and $D_2 = D_1 \cup \{\delta_4\}$, where the new rule is as follows:

$$\delta_4 = \frac{\langle \{T \to A\}, \emptyset \rangle : \langle \emptyset, \emptyset \rangle}{\langle \{\neg(P \wedge A)\}, \emptyset \rangle} \,.$$

The process tree for T_2 is presented in Fig. 6. There are 2 $S\&C$ leaves but they correspond to one and the same extension. According to this extension L is a proper part of P and P is discrete with A. The computed localization of P is denoted by $P\text{ubs}_3$ in Fig. 4. Similarly to the classic Default Logic, our approach is non-monotonic in a sense that in general changing any element of a default theory, results in an unpredictable change of extensions.

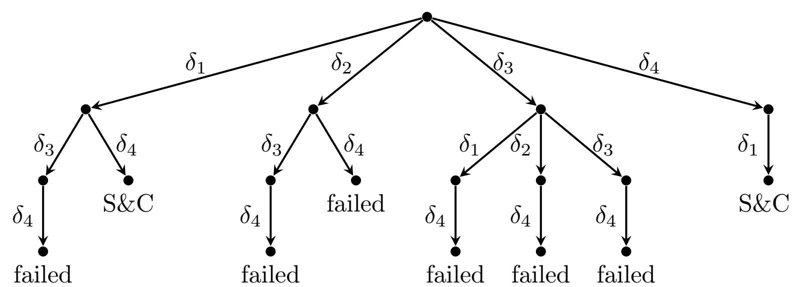

Fig. 6: A process tree for T_2.

6 Conclusions and Future Work

We have shown a formal method for default reasoning about topological relations. The method is decidable and we have provided an effective reasoning mechanism. Although our method is based on propositional logic, the provided application example to GIS shows that the method is expressive enough to represent interesting spatial configurations and may be used to solve some practical problems.

As a future work we consider determining method's computational complexity. It is not hard to show that its lower bound is Σ_2^P-hard for brave and Π_2^P-hard for cautious reasoning (e.g., by a reduction from $QBF_{2,\exists}$ similarly as in [8]). We suppose that the problems are Σ_2^P-complete and Π_2^P-complete but we have not proved the upper bound yet. We also consider increasing expressive power of the method but with computational complexity remaining as low as possible. In particular it is interesting how the approach may be extended in order to enable reasoning about other aspects of space such as distance, shape or orientation.

Acknowledgments. This research is partially supported by the Polish National Science Centre grant 2011/02/A/HS1/00395.

References

1. Antoniou, G.: Nonmonotonic reasoning. Mit Press (1997)
2. Bennett, B.: Spatial reasoning with propositional logics. KR 94, 51–62 (1994)
3. Bhatt, M.: (Some) Default and non-monotonic aspects of qualitative spatial reasoning. In: AAAI-08 Technical Reports, Workshop on Spatial and Temporal Reasoning. pp. 1–6 (2008)
4. Cohn, A.G.: Qualitative spatial representation and reasoning techniques. In: KI-97: Advances in Artificial Intelligence. pp. 1–30. Springer (1997)
5. Delgrande, J.P., Schaub, T.: On the relation between reiter's default logic and its (major) variants. In: Symbolic and Quantitative Approaches to Reasoning with Uncertainty, pp. 452–463. Springer (2003)
6. Donlon, J., Forbus, K.D.: Using a geographic information system for qualitative spatial reasoning about trafficability. In: Proc. of the Qualitative Reasoning Workshop (1999)
7. Dubba, K., Bhatt, M., Dylla, F., Hogg, D.C., Cohn, A.G.: Interleaved inductive-abductive reasoning for learning complex event models. In: Inductive Logic Programming, pp. 113–129. Springer (2012)
8. Gottlob, G.: Complexity results for nonmonotonic logics. Journal of Logic and Computation 2(3), 397–425 (1992)
9. Guesgen, H.W., Albrecht, J.: Imprecise reasoning in geographic information systems. Fuzzy Sets and Systems 113(1), 121–131 (2000)
10. Hartley, R.T.: A uniform representation for time and space and their mutual constraints. Computers & Mathematics with Applications 23(6), 441–457 (1992)
11. Hazarika, S.M.: Qualitative spatial change: space-time histories and continuity. Ph.D. thesis, The University of Leeds (2005)
12. Möller, R., Wessel, M.: Terminological default reasoning about spatial information: A first step. In: Spatial Information Theory. Cognitive and Computational Foundations of Geographic Information Science, pp. 189–204. Springer (1999)
13. Mostowski, A.: Thirty years of fundational studies. Blackwell (1996)
14. Randell, D.A., Cui, Z., Cohn, A.G.: A spatial logic based on regions and connection. KR 92, 165–176 (1992)
15. Reiter, R.: A logic for default reasoning. Artificial intelligence 13(1), 81–132 (1980)
16. Ryzko, D., Rybinski, H.: Distributed default logic for multi-agent system. In: Proceedings of the IEEE/WIC/ACM international conference on Intelligent Agent Technology. pp. 204–210. IEEE Computer Society (2006)
17. Shanahan, M.: Default reasoning about spatial occupancy. Artificial Intelligence 74(1), 147–163 (1995)
18. Tarski, A.: Sentential calculus and topology. In: Logic, Semantics, Mathematics. Oxford Cladendon Press (1956)
19. Wheeler, G.R., Damásio, C.: An implementation of statistical default logic. In: Logics in artificial intelligence, pp. 121–133. Springer (2004)
20. Yao, X., Thill, J.C.: Spatial queries with qualitative locations in spatial information systems. Computers, environment and urban systems 30(4), 485–502 (2006)
21. Zhou, Y., Lin, F., Zhang, Y.: General default logic. In: Logic Programming and Nonmonotonic Reasoning, pp. 241–253. Springer (2007)