General Introduction

Insurance portfolio management (IPM) is the process of managing multiple insurance policies (across potentially many overlapping domains) to the end of minimizing insurance costs while ensuring adequate coverage. By studying the insurance policies corresponding to a suite of policies, we can analyze a portfolio of insurance to ensure adequate coverage and avoid overlapping/redundant policies. Current IPM processes are human-driven, consisting of a lawyer or the policyholder examining insurance policies written in legalese/natural language to determine whether the policies cover certain instances/classes of claims. Such human-driven IPM is expensive in terms of time and human labor, while providing no guarantees as to its consistency across analyses or its accuracy. Thus, we believe IPM would benefit greatly from automation. To this end, we propose representing insurance policies as logic programs.

Such “computable insurance policies” immediately admit automated *claims analysis*, in which specific situations can be tested for compliance with an insurance policy’s terms and conditions. Furthermore, representing computable insurance policies as logic programs permits comprehensive *insurance portfolio analysis* via containment testing, in which a client’s existing insurance portfolio can be tested against to determine whether the client’s needs are met and whether additional insurance would be redundant. Such containment testing amounts to determining whether entire classes of situations are covered in a sound and complete way, a task which is *highly impractical* in existing computable insurance policies.

We first present an introduction to logic programming, followed by a discussion of how and why to represent insurance policies as logic programs, and end with a presentation of containment testing, where we highlight (i) how to apply containment testing to enable automated IPM and (ii) existing algorithms for performing containment testing.

Logic Programming Introduction

Logic programming is a form of declarative programming in which statements take the form of sentences in symbolic logic. In this paper, we will consider a specific but emblematic logic programming language called Epilog to ground the syntax of our examples. A logic program represents knowledge about the world and consists of a collection of facts called a dataset and a collection of rules called a ruleset.

[Definition of Constants]

The collection of constants in a logic program comprises the vocabulary of that program.
There are three types of constants…
[symbols/object constants]
[constructors]
[predicates]

A fact is an expression formed from an n-ary predicate and n ground terms. The semantics of the dataset are that all facts present in the dataset are assumed to be true, and all facts absent from the dataset are assumed to be false. An atom is a fact except its terms need not be ground (i.e. they can contain variables). A literal is either an atom or the negation of an atom. [The closed-world assumption, feeds into negation as failure]

A rule is in essence a reverse implication - it consists of an n-ary head with a predicate and n terms, and a body consisting of a conjunction of some finite number of subgoals (positive or negative literals). We say that the head of a rule is true if each of the subgoals in its body are true, for some consistent replacement of the variables in both. A negative subgoal is true as long as there is some consistent replacement of the variables in it that yields a fact that cannot be found/derived in the logic program. The heads of multiple rules can have the same predicate, and all rules with the same predicate in their head together comprise a view definition.

Querying a logic program then looks like writing some positive literal (with or without variables) with a base or view predicate (or defining one or more new views, then using one of the new view predicates) and asking the program which ground instances of that literal are derivable in the logic program. The set of all matching ground instances comprise the answer to the query.

Insurance policies as Logic Programs

Why would we want to represent an insurance policy as a logic program, rather than as a program written in an imperative language?
Logic programs can represent the logical flow of the contract more naturally than imperative languages, they are inherently more explainable/human-readable/ interpretable, formal verification is easier to perform on them, and, as we shall see, we can perform containment testing on them, something that is *highly impractical* in imperative programming approaches.

How to represent insurance policies as logic programs
Representing an insurance policy as a logic program looks like formulating an ontology and set of rules such that we can query the computable contract to answer questions about coverage. Such coverage queries can take many forms, such as "should this insurance claim be paid or declined?"(The binary coverage problem.) "If this claim is to be paid, how much should be paid out?" (The quantity coverage problem.) "Is any claim with the described characteristics covered by this contract?" (The needs/redundant coverage problem, the answering of which is facilitated by containment testing.)

One possible approach to representing insurance policies as logic programs is to have one view predicate indicate coverage, and instances of it indicate coverage for a specific claim e.g. covered(claim1). The absence of coverage for a specific claim would then look like the absence of covered(C) for a claim C, (denoted ~covered(C)). The base relations would represent the input to the contract: all information necessary to make a coverage determination, excluding rules specific to an insurance policy. Such information may include

- which person holds which policy [policy.holder(policy1, person1)]
- information about each person
    - date of birth [person.dob(person1, 1980_08_14)]
    - preexisting conditions [person.preexisting_condition(person1,asthma)]
- the relationships between people [person.parent(person2, person1)]
- the locations of each hospital [hospital.country(hospital1, canada)]
- the details of a specific claim
    - which policy a claim was filed under [claim.policy(claim1, policy1)]
    - which hospitalization a claim is filed regarding [claim.hospitalization(claim1, hospitalization1)]
    - the incident the claim is regarding [claim.incident(claim1, incident1)]
    - which exclusion applies, if any [claim.exclusion(claim1,skiing)]
- the details of a specific incident

        - the type of incident [incident.type(incident1, hospitalization), incident.type(incident1, car_crash)]
        - who was hospitalized [hospitalization.patient(hospitalization1,person2)]
        - when it began [hospitalization.startdate(hospitalization1, 2022_06_01), hospitalization.starttime(hospitalization1, “14:30”),]
        - the condition that caused the hospitalization [hospitalization.cause(hospitalization1, skiing_injury)]

And the rules of the logic program would represent the policy wording which determines which claims will be covered by the policy. The coverage predicate would then be defined in terms of one or more rules, each of which is defined in terms of base relations and/or view relations.
Such rules may look like:

- Recommend that a health insurance claim be paid if the policy under which the claim was filed is in effect, the hospitalization lasted at least a full day, and no exclusions apply.
    - covered(C) :-
```
          policy_in_effect(C) &
          full_day(C) &
          ~exclusion_applies(C)
```
- A plan is in effect during the period of the claim if the hospitalization that is listed in the claim began and ended while the policy was in effect.
    - policy_in_effect(C) :-
```
           claim.policy(C,P) &
           claim.hospitalization(C,Z) &
           policy.startdate(P,PS) &
           policy.enddate(P,PE) &
           hospitalization.startdate(Z,ZS) &
           hospitalization.enddate(Z,ZE) &
           overlap(PS,PE,ZS,ZE)
```

- The hospitalization listed in a claim lasted at least a full day if its duration was at least 24 hours.
    - full_day(C) :-
```
          claim.hospitalization(C,Z) &
           duration(Z,DURATION) &
           less(DURATION,86400000)
```

- The duration of a hospitalization is the difference between its start and end
    - duration(Z,DURATION) :-

    - hospitalization.startdate(Z,SD) &
      hospitalization.starttime(Z,ST) &
      hospitalization.enddate(Z,ED) &
      hospitalization.endtime(Z,ET) &
      timedifference(SD,ST,ES,ST)

- An exclusion applies if any exclusion applies.
    - exclusion_applies(C) :- exclusion_1a(C)
    - exclusion_applies(C) :- exclusion_4.3m(C)
    - …

- Exclusion 1a applies if the cause of the hospitalization was skiing
    - exclusion_1a(C) :-
      claim.hospitalization(C,Z) &
      hospitalization.cause(Z,skiing)
- Exclusion 4.3m applies if the cause of the hospitalization was a preexisting condition
    - exclusion_4.3m(C) :-
      claim.hospitalization(C,Z) &
      hospitalization.patent(Z,PERSON) &
      hospitalization.cause(Z, CAUSE) &
      person.preexisting_condition(PERSON,CAUSE)

[Example of how a query would work?]

We can then represent an entire insurance portfolio as the union of the logic programs that represent the insurance products owned(?) by the client, assuming no interactions between insurance products [unsure how much detail to give about the union process] [not currently sure of how to address/discuss the case of interactions between products].
Since an insurance claim is covered by a computable insurance portfolio iff at least one computable insurance policy in the portfolio would cover it, and a union of computable insurance policies indicates coverage for a claim iff at least one of the unioned computable insurance policies would indicate coverage, the union represents the complete coverage of the portfolio.

We note that a client's insurance needs can be represented as a logic program in a very similar [identical, so far - is it too strong to use that wording?] manner to how insurance policies are represented. Such a logic program would have the same view predicate indicating coverage, defined by rules written in terms of view predicates and the same set of base predicates. The program’s rules would be written to indicate coverage in exactly the situations matching the client’s needs.

[Examples, illustrating both general and highly-specific needs]

A collection of multiple needs can be represented similarly to how a collection of insurance policies is represented - as a union of logic programs.

Containment Testing

[Definition of database instance]

[Definition of minimal model]

Containment testing of logic programs is the process of determining whether one logic program will return all of the query answers generated by another. Formally, consider two logic programs Q_1 and Q_2 with the same query predicate and base predicates, and a database instance D in terms of those base predicates. The value of a logic program Q evaluated on a database D is denoted Q(D), and is a set of all atoms in the minimal model of Q U D that mention the query relation. We say that Q_1 is contained within Q_2 (denoted $Q_1 \leq Q_2$) iff for every database instance D, $Q_1(D)$ \subseteq $Q_2(D)$.

In the case of computable insurance policies, our query relation would be the view predicate indicating coverage. This means that, given computable insurance policies Q_1 and Q_2, if $Q_1 \leq Q_2$ then every claim that would be covered by Q_1 would also be covered by Q_2.

Note that, since we can represent clients' coverage needs as logic programs with the same query and base predicates as those used in computable insurance policies, *containment testing can be applied between policies and coverage needs as well*. That is, given a logic program Q_1 representing the coverage needs of a client, and a computable insurance policy Q_2, if $Q_1 \leq Q_2$ then the insurance policy covers all of the clients needs. [alt. every claim that a client would want to be covered is covered by the insurance policy]

Containment testing for Unions of logic programs

[To allow me to present the more straightforward proof in Ullman, I may restructure the presentation of computable insurance policies, their union, and the following paragraph.]

We can generalize containment testing between programs to testing between unions of programs. Since the process of taking the union of logic programs results in a single logic program $U_{i=1}^{n} Q_i$ and an insurance claim is covered by a computable insurance portfolio iff at least one computable insurance policy in the portfolio would cover it, we need only show that $(U_{i=1}^{n} Q_i)(D) = Q_1(D) \text{ U } Q_2(D) \text{ U } \dots \text{ U } Q_n(D)$. That is, we will show that the program resulting from the union returns coverage in exactly the same cases as the insurance portfolio would if its constituent policies were queried individually.

[Proof]

This shows that testing for containment against the union of the policies in an insurance portfolio allows automated IPM to test for needs coverage and redundant insurance.

Algorithms for containment testing

[Feels weird to wait this long to present algorithms, but I like the conceptual flow more with it down here. Now that I've thought through the rest of the paper more, will think more about how to include this section earlier.]

[Conjunctive queries and unions of them - frozen heads/canonical databases]
[CQ's with negations - Levy-Sagiv test]
[Discuss algorithms for CQ's with arithmetic?]
[If discussing arithmetic above, discuss limitations of taking unions for CQ's with arithmetic?]