Intelligent System - Expert System in Rust

Lars Grit

28 november 2025

1 Beschrijving en test van de werking

Mijn expertsysteem is geschreven in Rust als een simpel productie-regelmachine met een knowledge base die feiten als Person en Parent opslaat. Ik heb dit in Rust geschreven zodat ik een reden had om het te leren!

- `Person{name, gender}`: bijvoorbeeld `"lars", Male`, `"lotte", Female`, enz.
- `Parent{parent, child}`: bijvoorbeeld `Parent{arie,ellis}`.

Alle andere relaties zoals vader, moeder, broer of zus, oom, etc. Worden afgeleid door regels. Daarmee hou ik de initiële KB bewust zo klein mogelijk.

Voorbeeld van een testscenario (in `main`):

- Personen: Lars, Lotte, Ellis, Erik, Arie, Ariejan.
- Ouders:
  - Arie is ouder van Ellis en Ariejan.
  - Ellis en Erik zijn ouders van Lars en Lotte.

Na het uitvoeren van `infer_all(&mut kb)` print het programma o.a.:

- `Sons of erik: ["lars"]`
- `Sons of arie: [äriejan"]`
- `Grandsons of arie: ["lars"]`

De resultaten komen overeen met de familie boom (Lars is zoon van Erik, Ariejan is zoon van Arie)

2 Futureproof ontwerp

2.1 Eenoudergezinnen

Een oudergezin werdt al ondersteund: de KB bevat maar een Parent feit per kind (bijvoorbeeld alleen `Parent{ëllis","lars"}`).

De regels gebruiken alleen de ouder/child relatie, niet het aantal ouders per kind.

2.2 M/V/X en genderneutraal

De enumeratie `Gender` bevat naast `Male` en `Female` ook `X` voor genderneutraal / non-binaire personen. De afgeleide feiten `Father` en `Mother` worden alleen gemaakt als de persoon of `Male` of `Female` is. In de toekomst zou ik nog een gender neutrale term kunnen toevoegen.

2.3 Geslachtswijzigingen

In de huidige versie heeft elke persoon precies één `Gender` dat verandert kan worden.

2.4 Huwelijk tussen personen van hetzelfde geslacht

Het systeem werkt niet in termen van huwelijken, maar in termen van de relationele feiten `Parent{parent, child}` en `Person`. Daarom is een huwelijk tussen twee ouders van hetzelfde geslacht ondersteund.

2.5 Huwelijken met drie personen

Het systeem kan ook huwelijken met drie ouders aan er komen dan simpelweg drie `Parent` feiten per kind.

3 Forward chaining en backward chaining

3.1 Forward chaining

Het systeem past *forward chaining* toe via de functie `infer_all`. Daarin wordt in een lus herhaaldelijk een set regels aangeroepen:

```
fn infer_all(kb: &mut KnowledgeBase) {
    loop {
        let mut changed = false;
        changed |= infer_father_mother(kb);
        changed |= infer_sibling(kb);
        changed |= infer_uncle(kb);
        changed |= infer_cousin(kb);
        changed |= infer_nibling(kb);
        changed |= infer_grandson(kb);
        if !changed {
            break;
        }
    }
}
```

Elke `infer_*` functie leest bestaande feiten en voegt eventueel nieuwe afgeleide feiten toe aan de KB (bijvoorbeeld `Father`, `Sibling`, `Grandson`). Dit gaat door totdat er geen nieuwe feiten meer bijkomen: dan is de KB "saturated".

3.2 Backward chaining

In deze oefening is geen volledige backward chaining geimplementeerd maar er zijn wel query functions die doelgericht zoeken feiten:

```
fn sons_of(kb: &KnowledgeBase, parent_name: &str) -> Vec<&'static str> {
    kb.facts
      .iter()
      .filter_map(|f| {
          if let Fact::Father { father, child } = f {
              if *father == parent_name && kb.has_person(child, Gender::
                  Male) {
                  Some(*child)
              } else {
                  None
              }
          } else {
              None
          }
      })
      .collect()
}
```

3.3 Permanente vs. tijdelijke feiten

Afgeleide feiten (`Father`, `Mother`, `Sibling`, enz.) worden permanent aan de `KnowledgeBase` toegevoegd. Ze blijven in de hashset staan.

4 Inference engines in de praktijk

Het hier gebruikte systeem is een eigen, minimalistische implementatie in Rust en geen bestaande expert system shell. De code is wel gebaseerd op Experta die in de praktijk gebruikt wordt voor bijvoorbeeld configuratiesystemen.

Door het in Rust zelf te implementeren is het systeem wel klein maar niet geoptimaliseerd zoals bestaande oplossingen.

5 Aanpak, code en reflectie

5.1 Aanpak

De aanpak was als volgt:

1. Kiezen van een minimale set basisfeiten: `Person` en `Parent`.
2. Definiëren van afgeleide feiten (`Father`, `Sibling`, `Uncle`, `Cousin`, `Nibling`, `Grandson`).
3. Implementeren van forward chaining regels in Rust die over de bestaande feiten itereren en nieuwe feiten toevoegen tot er geen veranderingen meer zijn.
4. Schrijven van simpele queryfuncties (`sons_of`, `grandsons_of`) als lichte vorm van backward chaining.
5. Testen met een kleine familieboom gebaseerd op de eigen situatie.

5.2 Code

De kern van de code bestaat uit:

- de `Fact` enum voor alle typen feiten;
- de `KnowledgeBase` met een `HashSet<Fact>`;
- `infer_father_mother`, `infer_sibling`, `infer_uncle`, `infer_cousin`, `infer_nibling`, `infer_grandson`;
- queryfuncties zoals `sons_of` en `grandsons_of`.

5.3 Reflectie

Het bouwen van mijn expert systeem in rust was leuk maar wel erg uitdagend en heb meer gebruik gemaakt van outside sources dan dat ik had moeten doen voor een experta bijvoorbeeld dat veel werk voor je doet. Toch was het leuk om het te leren!

Het ontwerp is redelijk futureproof omdat:

- het aantal ouders per kind niet geforceerd is (dus eenouder en drieoudergezinnen zijn mogelijk),
- het geslacht expliciet als enum is gemodelleerd (M/V/X),

Een mogelijke vervolgstap zou zijn om een echte backward chaining laag toe te voegen.