Ghent Prolog

Tibo De Peuter

12 mei 2025

Samenvatting

Ghent Prolog is een command-line Prolog interpreter geschreven in Kotlin. Het biedt een subset van de functionaliteit van SWI-Prolog, waaronder database operaties en meta abstracties. Hoewel het programma nog niet als equivalent van SWI-Prolog kan worden beschouwd, is Ghent Prolog een nuttig leerproject.

1 Overzicht

Ghent Prolog is een proof of concept van een Prolog interpreter, als alternatief voor SWI-Prolog. Het volgt grotendeels ISO en ISO, 1995, de ISO Prolog standaard, maar is niet volledig compatibel.

Ghent Prolog werd geschreven in Kotlin en maakt gebruik van zowel object-gerichte als functionele concepten. Er werden twee bibliotheken gebruikt: `better-parse`, voor het parsen van een gegeven grammatica, en `kotlin-argparser`, voor het gebruik van command-line argumenten.

De verschillende fases van een interpreter (lexing, parsing, evaluatie, etc.) zijn in Ghent Prolog zo goed mogelijk gescheiden. Op die manier blijft de implementatie modulair en uitbreidbaar.

De volgende secties concentreren zich op de evaluatie-fase van de interpreter. Voor meer informatie over de lexing en parsing, zie sectie A.

2 Programma en REPL

Ghent Prolog's evaluatiemethode kan gezien worden als een vorm van *backward chaining* (Russell en Norvig, 2016), maar komt niet exact overeen met de definitie.

Ghent Prolog maakt gebruik van functie-oproepen en streams om een backtracking mechanisme te implementeren. Bijzondere logica met betrekking tot *choicepoints* en foutenafhandeling maakt gebruik van de Kotlin `Result` klasse. Om de resultaten van unificatie en evaluatie *lazy* te verwerken, wordt er gebruik gemaakt van Kotlin `Sequence`'s (*lazy streams*).

Meer specifiek wordt er gebruik gemaakt van de volgende types:

```
typealias Substitutions = Map<Term, Term>
typealias Answer = Result<Substitutions>
typealias Answers = Sequence<Answer>
```

Een lege `Sequence` betekent dat er geen oplossingen zijn gevonden. Een `Result.success` met `Substitutions` representeert een oplossing met de resulterende substituties (die leeg kan zijn). Een `Result.failure` betekent een fout of bijzondere logica die afgehandeld moet worden.

Er werden twee interfaces gedefinieerd:

```
interface Resolvent {
    fun solve(goal: Goal, substitutions: Substitutions): Answers
}

interface Satisfiable {
    fun satisfy(substitutions: Substitutions): Answers
}
```

Klassen die `Resolvent` implementeren kunnen een *goal* oplossen, bijvoorbeeld een database of een regel (*rule*). Klassen die `Satisfiable` implementeren kunnen worden geëvalueerd, bijvoorbeeld operatoren.

De verschillende Prolog termen werden voorgesteld als klassen op basis van SWI-Prolog's Glossary of Terms, met bijkomende inspiratie uit Deransart e.a., 1996.

3 Evaluatiestrategie

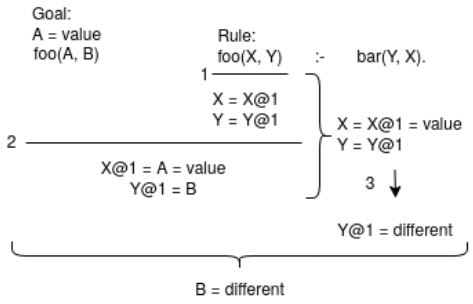
Ghent Prolog maakt gebruik van een depth-first zoekstrategie.

Op het moment dat de databank een goal gevraagd wordt (*query*), worden de volgende stappen doorlopen:

1. De databank geeft de goal beurtelings door aan de overeenkomende clauses, aan de hand van `solve`.
2. Voor elke clause wordt de goal met de head geünificeerd, wat nieuwe substituties kan introduceren.
3. De body van de clause wordt geëvalueerd (`Body.satisfy`), die zo de nieuwe goal wordt. Deze procedure wordt herhaald totdat de goal `true` of `false` is.
4. Het programma zal backtracken naar de functie die de laatste stap uitvoerde, en de pas geïntroduceerde substituties omhoog doorgeven. Daar wordt de juiste logica uitgevoerd, of het resultaat wordt verder doorgegeven aan de volgende stap.

Doorheen dit proces kunnen variabelen hernoemd worden aan de hand van `numbervars/1`. Een schematisch overzicht van dit proces is te zien in figuur 1.

Operator-specifieke logica bevindt zich in de klassen van de operatoren zelf.



Figuur 1: Voorbeeld van een substitutie met variable renaming.

In de volgende subsecties worden de belangrijkste onderdelen van de evaluatiestrategie in meer detail besproken.

3.1 Unificatie

Het unificatie-algoritme is gebaseerd op Robinsons Unificatie-algoritme, zoals beschreven door Boizumault, 1993, met inspiratie van Russell en Norvig, 2016. Merk op dat er gebruik gemaakt wordt van de *occurs check*.

De broncode voor het algoritme kan teruggevonden worden in `prolog/logic/unification.kt`.

3.2 Cut

De cut operator geeft altijd een `Result.failure(AppliedCut)` terug wanneer `satisfy` wordt opgeroepen. Deze uitzondering moet dan afgehandeld worden door de aanroepende functie, wat meestal resulteert in het teruggeven van de eerstvolgende oplossing en het onderbreken van de logica.

3.3 Meta abstracties

Ghent Prolog ondersteunt meta abstracts zoals `functor/3`, `arg/3`, `=../2` en `clause/2`. Daarnaast werden ook de operatoren `reset/3` en `shift/1` ingebouwd. Die laten toe om *delimited continuations* te gebruiken.

De ingebouwde ondersteuning van delimited continuations maakt opnieuw gebruik vaan een uitzondering. Deze keer heet die `Result.failure(AppliedShift)`, die de nieuwe substituties, doorgespeeld term (*ball*) en de nieuwe *continuatie*. Net zoals bij de cut operator, moet deze uitzondering afgehandeld worden door de aanroepende functie. In dit geval zal de `AppliedShift` steeds doorgespeeld worden, totdat de kotlinReset de uitzondering opvangt. Daarna gaat het programma verder met de nieuwe substituties en continuatie.

De broncode voor de meta abstracties kan teruggevonden worden in `prolog/builtins/delimitedContinuationsOperators.kt`.

4 Resultaat

De implementatie van Ghent Prolog ondersteunt de gevraagde functionaliteit, waaronder database operaties en meta abstracties. De architecturele verschillen met SWI-Prolog zijn echter groot, wat zowel positieve als negatieve gevolgen heeft.

De architectuur van Ghent Prolog is modulair en uitbreidbaar. Eens de evaluatiestrategie en datastructuren zijn gedefinieerd, is het eenvoudig om extra ingebouwde operatoren toe te voegen. Een volledig overzicht van de geïmplementeerde predicaten, waaronder extra's en uitbreidingen, kan teruggevonden worden in sectie D.

Code wordt snel onoverzichtelijk door het gebruik van `Sequence` en `Result`. Belangrijke logica zit genest in onduidelijke boilerplate. De code in 1 komt bijvoorbeeld in de meeste termen voor, weliswaar in verschillende vorm.

```
/* Function entry logic */
unifyLazy(a, b, subs).forEach { firstResult ->
    firstResult.map { firstSubs ->
        /* First stage logic, e.g. preparing the body etc. */
        c.satsify(firstSubs).forEach { secondResult ->
            secondResult.fold(
                onSuccess = { secondSubs ->
                    /* End result logic */
                    yield(Result.success(firstSubs + secondSubs))
                },
                onFailure = { failure ->
                    when (failure) {
                        is AppliedCut   -> /* Cut logic */
                        is AppliedShift -> /* Shift logic */
} } ) } } }
```

Listing 1: Voorbeeld van geneste boilerplate code

De implementatie bevat boilerplate code door het gebruik van overerving en interfaces in de klassenrepresentatie van termen. Deze methode wordt gebruikt om de onderlinge relaties tussen de verschillende termen te beschrijven. Hoewel nuttig voor de representatie van de AST en generieke functies, moeten de meeste klassen de `satsify` en `solve` methoden overschrijven. Dit zorgt voor boilerplate code en verspreidt de logica over verschillende klassen. Daarnaast kan het leiden tot verlies van type-informatie wanneer generieke functies en type-specifieke functies worden gemengd.

4.1 Functionele afwijkingen van SWI-Prolog

De huidige versie van Ghent Prolog heeft functionele afwijkingen van SWI-Prolog:

- Door het gebruik van de *occurs check* is het niet mogelijk om een oplossing te vinden voor recursieve unificatie. Voor `?- X = f(X).` vindt SWI-Prolog de oplossing `X = f(f(X))`, maar Ghent Prolog geeft `false` terug.
- Ghent Prolog maakt geen gebruik van SLD-resolutie, in tegenstelling tot ISO Prolog en SWI-Prolog (Toman, 2008).
- In de REPL wordt er niet meteen teruggekeerd naar de prompt als een query maar één oplossing heeft. Dit is een artifact om oplossingen die de database aanpassen, bijvoorbeeld `retract/1`, pas uit te voeren wanneer de gebruiker dat expliciet vraagt. Het gebruik van een iterator en `it.hasNext()` zou al de volgende oplossing genereren.

  Verder wordt ook de puntkomma (`;`) pas herkend na een newline.
- Voorlopig is het niet mogelijk om ingebouwde operatoren te overschrijven met `assert/1` of `retract/1`. Dit is een gevolg van het gebruik van de preprocessor.

5 Toekomstig werk

Bij de ontwikkeling van een variant van Ghent Prolog, wordt er best aandacht besteed aan de volgende zaken:

- Het gebruik van een stack voor choicepoints en andere informatie om het gebruik van exceptions weg te werken en eenvoudiger choicepoints te kunnen manipuleren. Zo kan bijvoorbeeld de cut operator transparanter *committen* tot de huidige oplossing door in één stap de juiste choicepoints te verwijderen, zonder dat elke operator een `AppliedCut` moet afhandelen.
- Het gebruik van een visitor-patroon in plaats van overerving, om boilerplate te verminderen, zoals aangegeven in sectie 4.

6 Conclusie

Ghent Prolog is duidelijk een proof of concept project. Het kan nog lang niet als equivalent van SWI-Prolog beschouwd worden.

Het implementeren ervan en het onderzoeken van de (voorlopig) ingebouwde functionaliteit biedt echter een goed inzicht in de werking van Prolog. Het geeft een beter begrip van de ingebouwde operatoren en hoe die interageren.

Persoonlijk vond ik de moeilijkste stap het opzetten van de evaluatie-strategie. Daarna ging het toevoegen van extra operatoren en functionaliteit vrij vlot. Ik ben ondertussen bezig met het zoeken van een manier om een bronbestand te schrijven waarmee de som van de getallen in een interval berekend wordt, aan de hand van `read/1`, `between/3` en delimited continuations.

Referenties

Boizumault, P. (1993, december 31). *The Implementation of Prolog*. Princeton University Press. https://doi.org/10.1515/9781400863440

Deransart, P., Ed-Dbali, A., & Cervoni, L. (1996, januari 1). *Prolog: The Standard*. https://doi.org/10.1007/978-3-642-61411-8

ISO, I., & ISO, I. (1995). IEC 13211-1: 1995: Information Technology—Programming Languages—Prolog—Part 1: General Core. *ISO: Geneva, Switzerland*, 1–199.

Russell, S., & Norvig, P. (2016). *Chapter 9. Inference in First-Order Logic*. Pearson Deutschland. https://elibrary.pearson.de/book/99.150005/9781292153971

Toman, D. (2008). *Chapter 9. SLD-Resolution And Logic Programming (PROLOG)*. University of Waterloo. https://cs.uwaterloo.ca/~david/cl/

Wikipedia contributors. (2025). Comparison of Prolog implementations — Wikipedia, The Free Encyclopedia [[Online; accessed 12-May-2025]].

A Lexing, parsing en preprocessing

Traditioneel zijn de lexer en parser aparte componenten. Om tijd te besparen werd echter voor het lexen en parsen van Prolog broncode en REPL-invoer gebruik gemaakt van de `better-parse` bibliotheek. Een gecompartimentaliseerde, onafgewerkt variant wordt besproken in sectie A.1.

De parser maakt gebruik van een vereenvoudigde Prolog grammatica, gebaseerd op de ANTLR Prolog grammatica, SICStus Prolog Full Prolog Syntax en simonkrenger/PrologParser's BNF. De uiteindelijk gebruikte grammatica kan teruggevonden worden in de `parser/grammars`.

Omdat de lexer en parser niet op maat gemaakt zijn, worden ze bewust zo eenvoudig mogelijk gehouden. Hun verantwoordelijkheid is het omzetten van inputtekst naar een basis Abstract Syntax Tree (AST), die enkel uit generieke Prolog termen bestaat. Merk op dat de parser ook verantwoordelijk is voor operator precedentie en associativiteit, zoals besproken wordt in sectie B.1.

Daarna worden de termen in de AST door de preprocessor omgezet naar specifieke klassen die de ingebouwde operatoren voorstellen. Termen worden herkend aan de hand van hun functor en ariteit.

Na de preprocessor is de AST klaar om gebruikt te worden door de evaluatie-fase.

A.1 Onafgewerkte Lexer en Parser implementatie

Origineel werden de lexer en parser op maat gemaakt, zonder het gebruik van een parser library. De lexer en parser waren gebaseerd op Crafting Interpreters, Robert Nystrom en Building a Parser from scratch, Dmitry Soshnikov. De parser was een recursive descent parser.

De voorlopige implementatie kan nog steeds teruggevonden worden op commit d5632e9.

B Aanvullende opmerkingen

B.1 Operator precedentie en associativiteit

Ghent Prolog ondersteunt operator precedentie en associativiteit. Deze functionaliteit bevindt zich in de parser, omdat de argumenten van een operator steeds rechtstreeks als parameters in de constructor van de klasse worden meegegeven.

Operator precedentie en associativiteit werd geïmplementeerd volgens de SWI-Prolog documentatie.

B.2 Database bewerkingen

Ghent Prolog laat toe om tijdens de programma-uitvoering database bestanden te laden met behulp van de `consult/1` predicaat.

De ingeladen predicaten worden als `static` gemarkeerd, maar kunnen `dynamic` worden met behulp van het `dynamic/1` predicaat. Dit is analoog aan SWI-Prolog.

B.3 Test driven development

Doorheen de ontwikkeling van grote delen van mijn implementatie heb ik gebruik gemaakt van Test Driven Development, onder andere met behulp van Kotlin `TODO`.

C Uitvoeren en testen

Om Ghent Prolog op een Windows, Linux of MacOS uit te voeren is het voldoende om Java te installeren en Ghent Prolog op te roepen met `./src/gpl`. De nodige stappen, waaronder het bouwen van een JAR, worden dan automatisch uitgevoerd.

De ingediende JAR kan ook handmatig opgeroepen worden met `java -jar ./build/gpl.jar`.

C.1 Testen

De testen kunnen uitgevoerd worden door de meeste IDE's.

Alternatief kunnen de testen uitgevoerd worden met `./gradlew test`.

D Overzicht van geïmplementeerde predicaten

Deze sectie geeft een overzicht van de geïmplementeerde predicaten in Ghent Prolog, gesorteerd volgens de categorieën die in de SWI-Prolog documentatie gebruikt worden.

Analysing and Constructing Terms

- `functor/3`
- `arg/3`
- `=../2`
- `numbervars/1`
- `numbervars/3`

Arithmetic

- `between/3`
- `succ/2`
- `plus/3`
- `=\=/2`
- `=:=/2`
- `is/2`
- `-/1`
- `+/1`
- `+/2`
- `*/2`
- `//2`
- `inf/0`

Comparison and Unification of Terms

- `=/2`
- `\=/2`
- `==/2`
- `\==/2`

Control Predicates

- `fail/0`
- `false/0`
- `true/0`
- `!/0`
- `,/2`
- `;/2`
- `|/2`
- `\+/1`

Database

- `retract/1`
- `retractall/1`
- `asserta/1`
- `assertz/1`
- `assert/1`

Declaring predicate properties

- `dynamic/1`

Delimited continuations

- `reset/3`
- `shift/1`

Examining the program

- `clause/2`

Forall

- `forall/2`

Loading Prolog source files

- `consult/1`
- `initialization/1`

Meta-Call Predicates

- `call/1`
- `once/1`
- `ignore/1`

Primitive character I/O

- `nl/0`

Term reading and writing

- `write/1`
- `writeln/1`
- `read/1`

Verify Type of a Term

- `var/1`
- `nonvar/1`
- `atom/1`
- `compound/1`

E Bestaande Prolog implementaties

Tijdens mijn onderzoek naar Prolog implementaties, kwam ik verschillende bestaande Prolog implementaties tegen die niet vermeld worden in Wikipedia contributors, 2025. Daarom zou ik durven stellen dat er geen gebrek is aan Prolog implementaties, maar dat er een gebrek is aan *goede* Prolog implementaties. Vaak zijn deze implementaties niet meer dan een proof of concept. Ze waren enkel nuttig als inspiratie om specifieke problemen die zich tijdens de ontwikkeling van Ghent Prolog voordeden op te lossen.

Voorbeelden van zulke implementaties zijn:

- adamjstewart/prolog, in OCaml
- miniprolog, in OCaml
- benjamin-hodgson/Amateurlog, in C#
- fmidue/prolog, in Haskell

- patirasam/Prolog-Interpreter, in Python

Van de implementaties die wel vermeld worden in Wikipedia contributors, 2025, zijn SWI-Prolog, SICStus Prolog en Strawberry Prolog de meest interessante.

F Dankwoord

Bedankt, LogicalCaptain om steeds nuttige informatie en voorbeelden te geven bij SWI-Prolog documentatie die iets minder duidelijk is. Uw nota's waren verhelderend en zorgden voor een beter begrip van en voor de nuances van SWI-Prolog.