Technische Hochschule Ingolstadt
Fakultät Informatik
Studiengang Informatik

Bachelorarbeit

OPS5 ähnliche Implementierung mit einem Retenetzwerk: Benutzung, Verhalten des Algorithmus, Laufzeit- und Speichereigenschaften

Lukas Hermann Frank

ausgegeben am: 06. Dezember 2023
abgegeben am: 12. Januar 2024

Erstprüfer: *Prof. Dr. rer. nat. Johann Schweiger*
Zweitprüfer: *Prof. Dr. rer. nat. Wolf-Dieter Tiedemann*

All rational action is in the first place individual action. Only the individual thinks. Only the individual reasons. Only the individual acts.

Ludwig Heinrich Edler von Mises

Zusammenfassung

In dieser Arbeit wurde ein OPS5 ähnlicher Interpreter in Java 21 implementiert, genannt **MemoRete OPS5**. Der Name setzt sich zusammen aus dem Begriff Memo wie Memoisation (oder englisch Memoization), Rete vom Rete-Algorithmus und OPS5 aus Official Production System 5. Auf dem Weg zur Implementierung wurden Vor- und Nachteile unterschiedlicher Ansätze besprochen, besonders auf die Indizierung von Teilmatches in Beta-Knoten wurde eingegangen. Es wurde gezeigt wie man den herkömmlichen Rete-Algorithmus modifizieren und verbessern kann. Des Weiteren wurde die Aktion Call von OPS5 für objektorientiertes Java angepasst. Der Code von MemoRete OPS5 wurde auf Github [Fra24] veröffentlicht.

Abstract

In this work, an OPS5-like interpreter was implemented in Java 21, called **MemoRete OPS5**. The name is made up of the term Memo such as Memoization, Rete from the Rete algorithm and OPS5 from Official Production System 5. On the way to implementation, preliminary and different approaches were discussed, in particular the indexing of partial matches in beta nodes was discussed. It was shown how the traditional rete algorithm can be modified and improved. Furthermore, the Call Action from OPS5 has been adapted for object-oriented Java. The code of MemoRete OPS5 has been published on Github [Fra24].

Inhaltsverzeichnis

1 Einleitung **5**

2 Arbeit Übersicht **7**
- 2.1 Ausgangslage . . . 7
- 2.2 Lösung . . . 7

3 Verwendung und Nutzen eines Produktionsregelsystems **8**
- 3.1 Vorteile . . . 8
- 3.2 Nachteile . . . 8
- 3.3 Weitere Untersuchungen: . . . 9
- 3.4 Produktionsregelsystem Programmerstellung . . . 11

4 Befehlssatz MemoRete OPS5 und Parser **12**
- 4.1 Überblick . . . 13
- 4.2 Strategie . . . 13
- 4.3 Literalisiere . . . 13
- 4.4 Startup . . . 13
- 4.5 Produktionsregel . . . 13
  - 4.5.1 Bedingungsteil . . . 14
- 4.6 Aktionsteil . . . 14
  - 4.6.1 Aktion Write . . . 14
  - 4.6.2 Aktion Make . . . 15
  - 4.6.3 Aktion Modify . . . 15
  - 4.6.4 Aktion Remove . . . 15
  - 4.6.5 Aktion Call . . . 16
  - 4.6.6 Funktion Compute . . . 17
  - 4.6.7 Aktion Halt . . . 18
- 4.7 Änderungen in MemoRete OPS5 . . . 18

5 Rete Algorithmus Implementierung **22**
- 5.1 Fakt (Fact) . . . 23
- 5.2 Rete Eintrag (ReteEntity) . . . 23
- 5.3 Wert (Value) . . . 25
- 5.4 Knoten, Propagierung und Strategie . . . 25
- 5.5 Alpha-Knoten . . . 26
- 5.6 Beta-Knoten . . . 28
  - 5.6.1 Existenz-Beta-Knoten ohne Atomvariable . . . 29
  - 5.6.2 Existenz-Beta-Knoten mit Atomvariable . . . 29
  - 5.6.3 Nichtexistenz-Beta-Knoten ohne Atomvariable . . . 29
  - 5.6.4 Nichtexistenz-Beta-Knoten mit Atomvariable . . . 30
  - 5.6.5 Multiset . . . 30

5.6.6 Multiset pro Wertetupel, Memoisation . . . 30
5.6.7 Viele-zu-viele Memoisation . . . 31
5.6.8 Wertetupel Nachschlag Matching . . . 32
5.6.9 Terminierungs-Knoten . . . 38
5.7 Aktionsteil . . . 38
5.7.1 Aktion Make, Modify, Remove . . . 38
5.7.2 Aktion Call . . . 38
5.7.3 Funktion Compute . . . 39

6 Schluss **40**

A Anlage digitale Medien **44**

Kapitel 1

Einleitung

Im Jahr 1979 hat der Informatiker Charles Forgy den Rete-Algorithmus entwickelt. Er ermöglicht das effizientete Abgleichen von Regeln in Produktionsregelsystemen. Seit dem hat sich Rete-Algorithmus in einer Vielzahl von BRMS's (business rules management systems) als Lösung bewährt, z.B. in Drools, JESS (Java Expert System Shell), CLIPS, IBM Operational Decision Manager. Später nach der Entwicklung des ersten Rete-Algorithmus hat Forgy gezeigt, dass es noch Potenzial für Verbesserungen gibt in Form von Rete-II, Rete-III und Rete-NT. Letzer soll sogar mindestens 500 mal schneller sein als der originale Rete-Algorithmus [Owe10].

Die neueren Retenetzwerke sind nicht quelloffen und so können die Verbesserungen hier nicht bestätigt werden. Generell gibt es aber, wie in den weiteren Kapiteln dieser Arbeit gezeigt wird, viele unterschiedliche Möglichkeiten den Rete-Algorithmus zu modifizieren, was zu unterschiedlichen Laufzeit- sowohl als auch Speichereigenschaften führt. Besonders Albert Mo Kim Cheng et al. (vgl. z.B. [CF00]) haben den Algorithmus auf Echtzeitfähigkeit optimiert. Der Fokus dieser Arbeit liegt aber auf einer generellen Implementierung, die zwar einen hohen Durchsatz haben soll, aber nicht unbedingt garantierte Antwortzeiten. Alleine die Verwendung von Hashmaps könnte im langsamsten Fall die Zugriffszeit von $O(1)$ auf $O(n)$ degradieren. Wer also Garantien zur Antwortzeit benötigt, dem sei zu Cheng et al.'s Arbeiten geraten.

Im Weiteren dieser Arbeit werden einige Unterschiedliche Variationen und Möglichkeiten vorgestellt, wie man den Algorithmus verbessern kann. Es werden Vor- und Nachteile erläutert und es wird auf ähnliche Probleme verwiesen. Man kann noch nicht sagen, dass der Rete-Algorithmus perfektioniert ist. Unterschiedliche Anwendungen stellen auch unterschiedliche Anforderungen an Laufzeit- und Speicherverhalten. Anwender müssen lernen was Ihnen am besten zur Lösung gereicht, denn es gibt noch keine allgemein beste Lösung.

Weiter wird immer wieder während der Arbeit auf den neu entwickelten Rete Algorithmus **MemoRete OPS5** eingegangen. Nur wenn man den Rete-Algorithmus selbst implementiert bemerkt man die ganzen Feinheiten und Details notwendig zur optimalen Funktion und dadurch die Möglichkeiten zur Verbesserung. Das Programm MemoRete OPS5 wurde in Java 21 entwickelt und es gibt die Möglichkeit zur anschaulichen Erklärung des Algorithmus. Wenn ein Konzept erklärt wird, dann wird auch auf die entsprechenden Stellen im Code verwiesen. Weiter ist zum Verständnis der Implementierung auch Bekanntschaft mit dem neuen OPS 5 ähnlichen Befehlssatz notwendig, dieser wird in Kapitel „Befehlssatz MemoRete OPS5 und Parser“ erklärt. Wie gesagt ähnelt dieser sehr dem herkömmlichen OPS5 und stimmt in großen Teilen mit ihm überein. Wer also einen der OPS5 Dialekte VAX-OPS5 oder LISP OPS5 kennt, sollte sich schnell zurechtfinden.

Um nur die wesentlichen Änderungen in MemoRete OPS5 zu sehen kann der Leser zum Kapitel „Rete Algorithmus Implementierung“ vorblättern.

Kapitel 2

Arbeit Übersicht

Als Hauptteil der Arbeit wurde das Java Programm MemoRete OPS5 erstellt. Hier gibt es eine kurze Übersicht.

2.1 Ausgangslage

Gegeben war ein kleines Java Programm zur Simulation von Robotern, enthalten in [Fra24, Dateipfad: MemoReteOPS5/src/main/ifs/ecar]. Ziel war die Implementierung eines OPS5 Interpreters basierend auf dem Rete Algorithmus, welcher auch auf die Methoden und Konstruktoren in der Klasse [Fra24, Dateipfad: MemoReteOPS5/src/main/ifs/ecar/ECar.java] zugreifen kann. Weiter war gefordert mit dem OPS5 Interpreter und der OPS5 Programmiersprache Roboter steuern zu können indem man auf die Klasse ECar aufsetzt.

2.2 Lösung

Es wurde ein OPS5 ähnlicher Interpreter implementiert, der mit der Call Aktion die Roboter steuern kann. Der Name der neuen OPS5 Implementierung ist MemoRete OPS5 [Fra24].

MemoRete OPS5 besteht aus einem Parser und dem Rete-Modell, das den Recognize-Act-Zyklus implementiert. Gesteuert wird der Interpreter zuerst beim Programmstart. Dort wird festgelegt welches OPS5 Programm geladen wird und ob man die interaktive Kommandozeile nutzen möchte [Fra24, Dateipfad: MemoReteOPS5/src/main]. Das Programm ist selbsterklärend, denn wenn man es ohne weitere Kommandozeilenparameter startet listet es die Optionen und Bedienungsweise auf. Es ist besser die Dokumentation des Programmverhaltens im Programmcode zu haben. So gibt es bei Änderungen nicht die Gefahr unterschiedlicher inkonsistenter Versionen der Dokumentation.

Wurde MemoRete OPS5 ohne interaktive Kommandozeile gestartet, so läuft das OPS5 Programm durch bis keine weiteren Regeln mehr gefeuert werden können oder bis die Aktion Halt aufgerufen wurde. Wurde der Interpreter stattdessen mit dem interaktiven Kommandozeilenmodus gestartet, so ist man nun in dieser Kommandozeile. Der Befehl „help“ zeigt einem die möglichen Befehle der interaktiven Kommandozeile (vgl. [Fra24, Dateipfad: MemoReteOPS5/src/main/ops5/Controller.java]).

Damit ist es möglich ein MemoRete OPS5 Programm zu schreiben und Roboter (oder auch allgemein Java Code) entweder einseitig oder interaktiv zu steuern.

Kapitel 3

Verwendung und Nutzen eines Produktionsregelsystems

Um die Verwendung und den Nutzen eines Produktionsregelsystems zu untersuchen sollen hier erst die potenziellen Vorteile genannt werden und sodann mögliche Einschränkungen. Gefolgt wird das durch weitere, zum Teil philosophische, Untersuchung.

3.1 Vorteile

Ein Produktionsregelsystem ermöglicht es Regeln als Wenn-Dann Regeln zu formulieren. Diese Wenn-Dann Regeln beziehen sich auf die Fakten einer Faktenbasis. Die Faktenbasis ist eingeteilt in unterschiedliche Kategorien von Fakten, welche Literale heißen. Ein Experte, also jemand der sich mit einer spezifischen Domäne gut auskennt, ist damit in der Lage sein Domänenwissen in Form von Literalen, Fakten und Wenn-Dann Beziehungen (in Form von Produktionsregeln) zu formulieren. Mithilfe dieser Formulierung kann der Experte sein Wissen in das Produktionsregelsystem abbilden. Das Produktionsregelsystem wiederum hat Schnittstellen zur Außenwelt, welche die Einträge in seiner Faktenbasis festlegt. Die Faktenbasis dient sowohl als Eingang von Informationen, als auch als Grundlage für Aktionen in der Außenwelt. Es folgt damit dem Eingabe-Verarbeitung-Ausgabe Prinzip eins Computers.

Ein Vorteil des Produktionsregelsystems, im Vergleich zu herkömmlichen Programmiersprachen ist, dass das Produktionsregelsystem besser das Wissen aus dem menschlichen Verstand nachbildet. Ein Mensch denkt eher in Wenn-Dann Beziehungen, z.B. „**Wenn** die Ampel Rot ist, **dann** muss ich stehen bleiben“ und „**damit** ich mit dem Auto stehen bleiben kann, **muss** ich die Bremse und die Kupplung drücken.“ Die Idee ist, dass auf diese Art ein Experte sein Expertenwissen möglichst vollständig und einfach abbilden kann.

3.2 Nachteile

Eine Programmiersprache steht und fällt an seiner Praxis. Es ist nicht einfach Expertenwissen auf ein Expertensystem abzubilden. Man könnte z.B. behaupten, dass Menschen generell Experten darin sind Sprache zu benutzen. Mühelos können sie Sätze formulieren, sie haben einen klaren Nutzen und Bedeutung. Will man aber die Fähigkeit zu Sprechen in ein Expertensystem speichern, sodass dieses auch sprechen kann, so wüsste man wahrscheinlich nicht wo man anfangen soll. Der Knackpunkt ist, dass man als Mensch und damit Experte im Sprachen sprechen, nicht Explizit weiß wie man Sprache benutzt. Es würden einem vielleicht eine Menge an

rudimentären Wenn-Dann Beziehungen, wie „Wenn die Sonne scheint, dann sag: ‚Das Wetter ist schön.'“ oder „Wenn ich mir den Zehen anhaue, dann sag ‚Aua!'“ einfallen. Das fängt aber noch lange nicht die Fülle und Dynamik an sprachlichen Äußerungen ein, die ein Mensch im täglichen Leben benutzt. Der Grund dafür könnte entweder sein, dass es einfach viele zu komplexe Bedingungen sind um praktisch Aufgelistet zu werden, oder dass die eigentlichen Gründe für Sprachaussagen im Unterbewussten des Menschen liegen und deswegen nicht Explizit genannt werden können. Selbiges gilt auch für Experten in ihren Domänen. Nur manche Teile vom Expertenwissen können in praktischer Weise auf ein Produktionsregelsystem abgebildet werden.

[VVP02, S.1 Übersetzt] gibt ein Beispiel zu einer konkreten Anwendung:

> „Das Expertensystem funktionierte so gut, dass es das Verhalten der Organisation veränderte – die Gruppe traf sich nicht mehr zum Zweck der Entwicklung der Verpackung, sondern traf sich stattdessen nur, um die Lösung des Expertensystems zu bestätigen und zu genehmigen. Trotz der klaren Vorteile, die mit der Verwendung eines Expertensystemansatzes verbunden sind, hörte die Gruppe auf, die Expertensystemanwendung zu verwenden, und das Verhalten der Gruppe kehrte zu Praktiken vor dem Expertensystem zurück.“

Sie schließen daraus, dass selbst „demonstrierbare Kosteneinsparungen nicht hinreichend zum Erfolg der neuen IT Anwendung [waren].“ und das System fest im Management und IT-Prozessen integriert sein müsste um erfolgreich zu sein [VVP02, S.78 Übersetzt].

3.3 Weitere Untersuchungen:

Das Schließen eines Expertensystems ist ähnlich dem deduktiven Prinzips der mittelalterlichen Scholastik. „Das typisch Scholastische war ein nahezu grenzenloses Vertrauen in die Macht und Zuverlässigkeit der Deduktion, des Schließens vom Allgemeinen auf das Besondere“ [Wik04]. Dies ähnelt dem schließen des Expertensystems. „Man nahm an, dass die fehlerfrei durchgeführte Deduktion zur Erkenntnis von allem vernunftmäßig Erkennbaren und zur Beseitigung aller Zweifel führen kann“ [Wik04]. Des Weiteren ähnelt ein Syllogismus in seinem Aufbau [Wik04] dem einer Produktionsregel.

Das heißt, dass die Produktionsregeln eines OPS5 Programms einem scholstischen Wissenskanon entsprechen. Die gegebenen konkreten Instanzen (Fakten) produzieren dann deduktiv alle vorhersagbaren Ergebnisse.

Die Festlegung, Findung und Neuerfindung von Begriffen, die dem Problem der Domäne spezifisch sind, ist ein wichtiger Teil der OPS5 Programmerstellung. Nur wenn man die richtigen Begriffe hat, kann man diese in Literale abfassen und hinreichende Produktionsregeln formulieren. Um eine Produktionsregel richtig zu formulieren ist es notwendig alle möglichen Ursachen zu finden die Einfluss auf die Wirkung haben könnten. Nur wenn man alle Ursachen erschöpfend erfasst hat, kann man sicher sein, dass die Produktionsregel, dann und nur und immer dann, feuert, wenn es angebracht ist. Jede Ursache muss also durch mindestens ein Literal erfasst werden. Wenn man die Literale und Produktionsregeln erfasst hat, kann man deren domänenspezifische Richtigkeit testen indem man es gegen alle möglichen Eingansfälle überprüft. Erkennt der Experte auch nur einen Fall der Unsinnig ist, muss die Wissensrepräsentation angepasst werden.

Bei Produktionsregeln besteht der Bedingungsteil aus den Prämissen oder Voraussetzungen, der Aktionsteil aus den Konklusionen oder Schlussfolgerungen. Es kann allerdings keine Deduktionen (Schluss vom allgemeinen auf etwas spezielles) oder Induktionen (Schluss vom speziellen (also ein Beispiel) auf das Allgemeine) unmittelbar in der Sprachsyntax abbilden. Produktionsregeln können nur über eine prozedurale Regel spezielles auf spezielles abbilden. Wenn man bei Produktionsregelsystemen von einer Wissensbasis spricht ist es also treffender zu sagen es handelt sich um eine Faktenbasis, denn Wissen schließt prozedurales Wissen und auch Verständniswissen mit ein. Wahres Verständniswissen ist auf den Menschen beschränkt, man kann nur jeweils das prozedurale Wissen abbilden. Je nachdem wie viel Verständniswissen, durch Explizitmachung, in prozedurales Wissen umgewandelt werden kann, kann Verständnis simuliert werden. Echte Lernfähigkeit scheint aber vorerst dem Menschen vorbehalten.

In Expertensystemen sind die Datentypen eindeutig bestimmt. Ein bloßer Tippfehler stellt schon ein absolutes Unterscheidungskriterium dar. Der menschliche Verstand verbindet aber ihm ähnlich Begriffe und Formulierungen, wie z.B. „Mensch“ und „vernunftbegabtes Lebewesen“. Gleichzeitig kategorisiert er, dass ein „Mensch“ zwar meist ein „vernunftbegabtes Lebewesen“, ein „vernunftbegabtes Lebewesen“ aber nicht zwangsläufig immer ein Mensch sein muss. So ist es also bei der OPS5 Programmerstellung notwendig Literale die in solcher kategorischer Beziehung zusammenhängen stets miteinander Konsistent zu halten. In der Liste von Fakten von „vernunftbegabtes Lebewesen“ müssen sich also stets mindestens alle Fakten von „Mensch“ befinden. Das gilt aber natürlich nur unter der Annahme, dass auch alle Menschen vernunftbegabt sind.

Schon Aristoteles argumentierte:

> „Die Argumente werden aus Prämissen gebildet, worüber aber die Deduktionen gebildet werden, das sind die Probleme. [Höf+09, S.38]“ „Daher überrascht es nicht, dass die Probleme und die Prämissen der Zahl nach gleich sind, denn aus jeder Prämisse lässt sich ein Problem bilden, indem man die Formulierung verändert. [Höf+09, S.39]“

Die Annahmen (Prämissen) sind also selbst nicht gesetzt. Die ersten Annahmen (initiale Faktenbasis) muss durch den Benutzer vorgegeben werden, sie sind also der Anfangspunkt, „die Welt“ für das System. Deren Richtigkeit und Angemessenheit kann nur der Benutzer wissen. Das System ist dazu nicht fähig. Das ist ein Unterschied zum Mensch, der in der Philosophie und auch anderen Denkschulen seine ersten Prämissen durchaus auch in Frage stellt. Es ist also anzunehmen, dass das System nur starr in seiner Problemdomäne verbleibt, aus sich selbst nicht dazu lernt und gleiche Fehler unter gleichen Bedingungen stets wiederholt. Dem System fehlt die Fähigkeit Gut von Schlecht zu unterscheiden. Es ist nur so gut wie seine Programmierung.

> „Das Allgemeine und auf alles Zutreffende dagegen kann nicht wahrgenommen werden, denn es ist kein Dieses und auch nicht jetzt; sonst wäre es nicht allgemein, denn was immer und überall ist, nennen wir allgemein [...] Wahrgenommen nämlich wird notwendig das Einzelne, das Wissen dagegen ist das Kennen des Allgemeinen. [Höf+09, S.79]“

Das System kann aus sich selbst nicht dazulernen. Jedes Lernen kann nur durch den Benutzer mit seiner Systemeinsicht mittels erstellen neuer Fakten, Literale und Produktionsregeln erfolgen. Ein dynamisches Lernen kommt durch Produktionsregeln, die selbst wiederum Produktionsregeln erstellen in Frage (mithilfe der „(build ...) “ Anweisung). Ob das aber praktisch für einen Benutzer anwendbar ist ist ungeklärt.

3.4 Produktionsregelsystem Programmerstellung

Abschließend wird eine kurze Anleitung zur Erstellung eines OPS5 Programms gezeigt:

1. Natürlichsprachliche Zielformulierung erstellen.
2. Ziele als Begriffe formulieren.
3. Anfangsbedingungen als Begriffe formulieren. Die Anfangsbedingungen schließen die gesamte zielrelevante Realität mit ein.
4. Zwischenziele als Begriffe formulieren, sodass Ketten von den Anfangsbedingungen zu allen Zielen möglich ist.
5. Diese Ketten konkret auflisten. Jede Kette ist eine Produktionsregel oder eine Menge von einander abhängigen Produktionsregeln.
6. Prüfen ob die Produktionsregeln hinreichend zur natürlichsprachlichen, allgemein verständlichen, Zielerreichung sind. Die Symbole und Deduktionen müssen explizit genannt werden und so übereinstimmen. Die Bedeutung jener ist dem Computer nicht bewusst. Nur der Mensch der ihn bedient weiß über seine Bedeutung und kann ein Urteil darüber fällen ob sie in dem Zusammenhang richtig dargestellt werden.

Kapitel 4

Befehlssatz MemoRete OPS5 und Parser

Soweit bekannt, ist OPS5 nicht standardisiert. Zur Wahl des Befehlssatzes wurde stattdessen das OPS5 Lehrbuch [Kri87] zur Hilfe genommen. Laut dem Lehrbuch [Kri87, S.1] „gab es [damals] als einzige Unterlage lediglich das knappe Handbuch von Charles L. Forgy als Bericht der Carnegie-Mellon Universität sowie einige kurze Veröffentlichungen anderer Universitäten.“ Weiter steht dort: „[d]ie Autoren konnten deshalb detaillierte Kenntnisse über OPS5 und dessen Möglichkeiten nur auf experimentellem Weg gewinnen.“ Es handelt sich also bei der OPS5 Sprachdefinition um eine die sich implizit am Programmverhalten definiert und nicht wie z.B. bei C um einem internationalen Standard.

Weiter verkompliziert wird eine einheitliche Sprachdefinition dadurch, dass es historisch zwei unterschiedliche OPS5 Dialekte gibt. [Kri87, S.28-29] nennen die herkömmliche LISP-OPS5 Variante, die Zugriff auf einen LISP-Interpreter besitzt und VAX-OPS5 (basierend auf der Programmiersprach BLISS), welches speziell für VAX Rechner der Digital Equipment Corporation (DEC) entwickelt wurde. Weiter steht dort, dass VAX-OPS5 diverse Erweiterungen in der Sprachdefinition im Vergleich zur LISP Version besitzt.

Für die Implementierung des MemoRete OPS5 Interpreters wurde VAX-OPS5 als Vorlage gewählt. Die wesentlichen Sprachelemente wurden beibehalten und es wurde auf die Java-Umgebung angepasst. Da die Grundstruktur gleich geblieben ist handelt es sich also um eine weitere Version der OPS5 Sprache.

Im folgenden wird kurz der neue MemoRete OPS5 Befehlssatz erklärt und wie dieser im Java Programmcode (speziell im Parser) umgesetzt ist. Abbildung 4.1 zeigt ein Codebeispiel, das die wesentlichen Elemente des MemoRete OPS5 Befehlssatzes enthält. Zur Übersicht wird dem Leser empfohlen das Dokument noch einmal in einem neuen Fenster zu öffnen um parallel den Erklärungen und Codeteilen folgen zu können. Zeilenangaben im Folgenden beziehen sich auch auf Codeteile aus der Abbildung 4.1. Verweise auf den Java Parser beziehen sich immer auf die Java Klassen im MemoRete OPS5 Programm [Fra24, Dateipfad: MemoRete-OPS5/src/main/ops5/parser] der Erstveröffentlichung.

Um das Beispielprogramm auszuführen kann man den Anweisungen der Readme Datei folgen [Fra24, Dateipfad: MemoReteOPS5/Readme.txt]. Folgendes sind die Kommandozeilebefehle um MemoRete OPS5 zu kompilieren und den Beispielcode aus Abbildung 4.1 auszuführen:

```
1 $ mvn clean package
2 $ java -jar target/MemoReteOPS5-0.0.1-SNAPSHOT.jar -f MemoReteOPS5lang.ops
```

4.1 Überblick

Überblickt man das Programm in Abbildung 4.1, so sieht man mehrere Codeblöcke die jeweils abgeschlossen mit Klammern `(...)` sind. Sie müssen einzeln geparst werden. Der Parser teilt den Code also in Blöcke auf. Realisiert ist das in der Methode:

```
public void parseOps5String(String str) throws Exception {...}
```

Sie separiert die Codeblöcke mithilfe eines Aufrufs der String Hilfsmethode:

```
1 res = StringHelpers.substringBetween(str, "(", ")")
```

welche intern einen Stapelspeicher an offenen Klammern pflegt um so festzustellen wann eine letzte Klammer, und somit ein Codeblock, schließt. Jetzt hat das Programm eine Liste von Codeblöcken. Je nachdem mit welcher Zeichenfolge der Block beginnt wird er unterschiedlich behandelt:

4.2 Strategie

Der Codeblock fängt mit `(strategy ...)` an, siehe Zeile 3 im Codebeispiel. Der Strategie Block definiert die Reihenfolge in der die Produktionsregeln feuern. Unterstützt werden die Strategien LEX und MEA. Sie funktionieren wie im Lehrbuch beschrieben [Kri87, S. 65-67], der entsprechende Code findet sich hier [Fra24, Dateipfad: MemoReteOPS5/src/main/ops5/Strategy.java].

4.3 Literalisiere

Der Codeblock fängt mit `(literalize ...)` an. Im Codebeispiel sieht man das mehrfach in den Zeilen 5, 13, 19 und 21. Literalize ist vergleichbar mit Java Klassen oder C Structs. Sie definieren die Datenklassen. Im Beispiel werden die Literale `robot`, `pose`, `init` und `cleanup` definiert. Innerhalb eines Literalize wird nach dem Namen der Datenklassen keine, ein oder mehrere Attribute definiert. `robot` hat die Attribute `name`, `type`, `velocity`, `location` und `instantiated`. `cleanup` dagegen hat keine Attribute definiert. Jedes Literal kann später als Fakt instantiiert werden, wobei jedes Attribut als Wert entweder `NIL`, ein `DOUBLE`, ein `INTEGER` oder einen `String` erhält. Der Wert der später konkret bei einem Attribut gespeichert ist ist in dem Java Record Value [Fra24, Dateipfad: MemoReteOPS5/src/main/ops5/workingmemory/data/Value.java] definiert.

4.4 Startup

Der Codeblock fängt mit `(startup ...)` an (siehe Zeile 23). Innerhalb der Startup-Anweisung kann man Fakten für den Programmstart instantiieren. Die einzelnen Anweisungen sind ähnlich der Aktion Make, blos dass keine Atomvariablen und keine Funktion Compute unterstützt wird. Die startup Anweisung ist optional und kann auch weggelassen werden.

4.5 Produktionsregel

Der Codeblock fängt mit `(p ...)` an, das kleine p steht für Produktionsregel. Produktionsregeln machen den Hauptteil des Programms aus (siehe Zeile 27 und dahinter). Produktionsregeln bestehen aus einem **Bedingungsteil** (auch IF oder Left Hand Side genannt) und einem **Aktionsteil** (auch THEN oder Right Hand Side genannt). Getrennt sind beide durch einen Pfeil „-->“. Wird der Bedingungsteil durch einen Kombination von Fakten erfüllt, so feuert die

Produktionsregel und die Aktionen des Aktionsteils werden ausgeführt. Des Weiteren können Produktionsregeln noch eine Liste von Atomvariablen und Elementvariable deklarieren, erkennbar an den spitzen Klammern `<>`. Produktionsregeln werden im Code in [Fra24, Dateipfad: MemoReteOPS5/src/main/ops5/workingmemory/data/ProductionRule.java] definiert.

4.5.1 Bedingungsteil

Der Bedingungsteil besteht aus einer oder mehreren Bedingungen. Eine Bedingung ist entweder eingeschlossen durch `()` oder `{}`. Ein Bedingung eingeschlossen mit geschweiften Klammern, definiert am Anfang eine Elementvariable (siehe z.B. Zeile 35), gefolgt von einer einfache Bedingung mit runden Klammern `()`. Die Fakten, die die innere einfache Bedingung erfüllen, belegen auch die Elementvariable. Eine Elementvariable verweist also immer auf einen Fakt. Elementvariablen können vor oder nach der einfachen Bedingung stehen.

Einfache Bedingungen beginnen immer mit dem Namen eines Literals, also z.B. `init` (Zeile 35) und danach mit keiner einer oder mehreren Unterbedingungen. Die einzelnen Unterbedingungen beginnen immer mit „`^`“ und einem Attributnamen gefolgt von einem Vergleichsoperator. Test auf:

- Gleichheit mithilfe von „`=`“, oder implizit indem man das „`=`“ weglässt. Fuktioniert mit jedem Datentyp
- Ungleichheit mithilfe von „`<>`“. Fuktioniert mit jedem Datentyp
- Größenvergleiche mit „`<, <=, >, >=`“ was nur mit `DOUBLE` und `INTEGER` funktioniert. Enthält ein Attribut `NIL` oder einen `String`, so wird der zugehörige Datentyp nicht weiter propagiert
- Auf gleichen Datentyp mit „`<=>`“, was natürlich mit jedem Datentyp funktioniert.

Gefolgt wird der Vergleichsoperator von einem konkretem Atom, wie `Integer 5`, `Double 5.0`, `String Teststring` oder `NIL`. Bedingungen können Elementvariablen und Atomvariablen belegen. Dazu folgendes Beispiel aus Zeile 35:

```
{<elemvariable_init> (init ^val 1 ^val <value>)}
```

Es handelt sich um eine Bedingung auf Fakten vom Typ `init`, welche im Attrubut `val` den Wert 1 gespeichert haben. Des Weiteren muss das Attribut `val` den gleichen Wert wie die Atomvariable `<value>` gespeichert haben. Fakten die diese Bedingungen erfüllen belegen dann die Elementvariable `<elemvariable_init>`.

4.6 Aktionsteil

Der Aktionsteil wird definiert durch eine oder mehrere Aktionen eingeschlossen durch runde Klammern „`()`“. Alle Aktionen inklusive der Funktion Compute befinden sich im Code in [Fra24, Dateipfad: MemoReteOPS5/src/main/ops5/workingmemory/data/action/].

4.6.1 Aktion Write

Aktionen Write starten mit `(write ...)`. Sie ermöglichen das schreiben zur Kommandozeile (stdout) für die Programmausgabe (siehe Zeilen 74, 88). Eine volle Aktion Write kann z.B. so aussehen:

```
(write |car | <carNr> | is bought by customer | <customerName> (CRLF)
  | for | (COMPUTE 3 + <atomVar4>) |oz gold| (CRLF))
```

Aktion Write kann folgende Teile in beliebiger Reihenfolge und Menge enthalten:

- Einen String, wie z.B. „|car |“, wobei nur der Teil zwischen den Trennzeichen „|“ gedruckt wird.
- Atomvariablen, wie z.B. <carNr>. Es druckt den Wert der in der Atomvariable <carNr> beim feuern der Produktionsregel gespeichert ist.
- Funktion Compute berechnet einen numerischen Wert und gibt diesen aus, genaueres findet sich in der Sektion „Funktion Compute“
- „(CRLF)“ druckt eine neue Zeile

4.6.2 Aktion Make

Aktion Make beginnt mit (make literalname123 ...), wobei literalname123 durch den konkreten Namen eines Literals erstetzt ist. Im Codebeispiel ist das also eines von den Literalen robot, pose, init oder cleanup. Danach definiert Aktion Make die Belegung der einzelnen Attribute. Wird ein Attribut nicht explizit belegt, so wird es implizit mit NIL belegt. Zur Attributbelegung gibt es folgende Möglichkeiten:

- Zuweisung eines Konkreten Werts, z.B. ^name SIM_DX1
- Zuweisung durch eine Atomvariable, z.B. ^name <na1>
- Zuweisung durch Compute, z.B. ^value (compute 2+5)

4.6.3 Aktion Modify

Aktion Modify startet mit (modify <elementVariable123> ...). Es modifiziert den Fakt, der in der gegebenen Elementvariable gespeichert ist, also in dem Beispiel die Elementvariable <elementVariable123>. Die definition der Werte der Attribute ist gleich wie in Aktion Make. Mit jeder Modifizierung wird der alte Fakt gelöscht und ein neuer mit modifizierten Werten erstellt. Aus diesem Grund kann auch eine Elementvariable nur einmal in einer Produktionsregel verwendet werden.

4.6.4 Aktion Remove

Aktion Remove besteht nur aus zwei Wörtern: (remove <elementVariable123>). Es löscht den Fakt der in der Elementvariable gespeichert ist, in diesem Fall also der Fakt in <elementVariable123>.

Remove unterstützt nicht die Addressierung eines Fakts nach Index, wie z.B. (remove 1). Das originale OPS5 wurde auf Basis von einem LISP Dialekt implementiert. Lisp unterstützt Atome und Listen und legt daher einen Programmiertyp nahe in dem konkret mit Indizes auf Listen Referenziert wird. Das in Java nachzubilden wäre aber umständlich und ineffizient, da Java neuere Sprachkonstrukte, wie Maps, usw. bietet. Die ausschließliche Verwendung von Elementvariablen ist stabiler und weniger Fehleranfällig in der Programmierung. Das neu Programmieren einer Produktionsregel kann damit nicht so leicht aus versehen zum falschen löschen von Fakten führen. Mit einer Elementvariable wird direkt auf die entsprechenden Fakten verwiesen anstatt über einen impliziten Verweis auf die Reihenfolge der Bedingungen.

In [Kri87, S. 48] wird das Resultatelement beschrieben: „[d]as Resultatelement dient als Speicher für Zwischenergebnisse.“ Weiter wird im Lehrbuch das Resultatelement als Liste dargestellt, welche die Ergebniswerte enthält. In MemoRete OPS5 ist es nicht notwendig Listen mit potenziell leeren Plätzen für die Verarbeitung im Aktionsteil zu unterhalten. Stattdessen werden direkt mit einer Map nur die Elemente gespeichert, die auch Daten beinhalten. Es vermeidet damit im Literal die Indizes für Attribute nachsehen zu müssen und kann stattdessen direkt die Attributnamen zur Adressierung verwenden.

4.6.5 Aktion Call

Aktion Call beginnt mit `(call ...)`. Aktion Call ermöglicht es über Nutzung von Java Reflections nahezu beliebige Java Programmteile aufzurufen. Es unterscheidet sich stark von der herkömmlichen Aktion Call, da es speziell für Java angepasst wurde. Intern enthält es eine Whitelist an erlaubten Java Pfaden um eine gewisse Grundsicherheit zu gewährleisten. Im Standard-Programm ist nur `"emo.ifs.ecar.ECar"` und eine unbedenkliche Klasse für Unit-Tests erlaubt. Möchte man auf andere Klassen zugreifen, muss man diese manuell im Code in [Fra24, Dateipfad: MemoReteOPS5/src/main/ops5/workingmemory/data/action/ActionCall.java] unter

```
static final private ImmutableList<String> allowedClasses =
    ImmutableList.of( ... );
```

hinzufügen.

Aktion Call ist in zwei Teile aufgeteilt mit der Trennung durch das Symbol „`=>`“. Der zweite Teil ist optional und wenn er nicht vorhanden ist muss auch das Trennungssymbol weggelassen werden.

Erster Teil: Call

Der Call Teil der Aktion Call kann im Beispielcode in Zeilen 39, 40, 50, 66, 99, 101 und 107 gefunden werden. Er ist aufgebaut nach folgendem Schema:

```
(call uniquename classpath.methodname ^attribute1 attributevalue1 ...)
```

`uniquename` ist ein einzigartiger Name, der entweder durch ein Atom, wie einen String, Double, Integer oder NIL festgelegt wird oder dynamisch durch eine Atomvariable, wie im Beispielcode in Zeile 99 usw. zu sehen. Der `uniquename` zusammen mit dem `classpath` definieren den Objektspeicherpfad für den Objektspeicher:

```
static final private HashBasedTable<String, String, Object>
   objectStore = HashBasedTable.create();
```

Bevor eine Klasse benutzt werden kann, muss sie mit `new` als `methodname` instanziiert und im Objektspeicher gespeichert worden sein (vgl. Beispielcode Zeile 99). Erst danach können die Methoden des Objekts aufgerufen werden, (vgl. Beispielcode Zeile 101). Mit diesem Aufbau des Objektspeichers ist es möglich beliebig viele Java Objekte von der selben Klasse, sowohl als auch Objekte von unterschiedlichen Klassen zu instantiieren und anzusprechen.

Die Attribute für den Call werden ähnlich wie in Aktion Make festgelegt. Die Besonderheit ist, dass auch Arrays vom Typ `Integer`, `int`, `Long`, `long`, `String`, `Float`, `float`, `Double` und `double` unterstützt werden. Einen Arrayparameter übergibt man indem man einen Stern „`*`“ nach dem Attributnamen anfügt und danach eine Liste von Parameterwerten:

```
(call callzo ops5.workingmemory.data.action.ActionCall.
   MockClassActionCallTest.new ^str * test1 test2))
```

Der Typ des Arrays wird automatisch ermittelt. Passen mehrere Konstruktoren oder Methoden zu den gegebenen Parametertypen, so wird der erste in der Liste gewählt (der erst deklarierte). Verwendet man Atomvariablen für den Aufruf, so kann der Typ der Atomvariable pro feuern der Regeln sich ändern. Für jeden solchen Aufruf wird zur Programmlaufzeit der passende überladene Konstruktor oder Methode gewählt. Eine mehrfache Verschachtelung von Arrays wird nicht unterstützt.

Zweiter Teil: Subaktion

Der zweite Teil ist optional, aber notwendig um die Rückgabewerte von Methodenaufrufen und Konstruktoren Primitiver Wrapper-Klassen (Integer, ...) im OPS5 Programm zu speichern. Bis auf die Fähigkeit Rückgabewerte mit „<?>“ zu benutzen, sind die Subaktionen in der Funktionsweise gleich den der entsprechenden Aktionen. Es gibt die Subaktionen Make, Modify und Remove wobei nur Make und Modify die Rückgabewerte mit „<?>“ verwenden können. Ein Beispiel:

```
1 (call <atom25> ops5.workingmemory.data.action.ActionCall.
     MockClassActionCallTest.test2 ^list * test1 test2
2 => modify <elemvar1> ^value3 <?> ^value <atom25>)
```

„<?>“ belegt das Attribut value3 mit dem Return Wert vom Aufruf der Methode test2:

```
public String[] test2(String[] list) { ... }
```

Handelte es sich beim Return Typ der Methode nur um einen String so würde nur der Wert von value3 mit einem String überschrieben. Da es sich aber um ein Array String[] handelt, wird je nach Arraylänge value3 und alle nachfolgenden Attributwerte mit den Arrayinhalten überschrieben. Ist die Arraylänge ungleich der Länge der verbleibenden Liste an Attributen, so stoppt das Überschreiben am Ende des jeweils kürzeren. Wenn das Return Array ein Attribut belegt und gleichzeitig eine explizite Zuweisung an das selbe Attribut vorhanden ist, so gilt immer die explizite Zuweisung vorrangig.

4.6.6 Funktion Compute

Funktion Compute beginnt mit (compute ...). Sie ist keine alleinstehende Aktion, sondern ist eine Funktion zur Berechnung eines Attributwerts, siehe z.B. Zeile 42 oder 52. Compute ist möglich innerhalb der Aktionen und Subaktionen Make und Modify. Evaluiert wird sie immer von links nach rechts, außer wenn Klammern gesetzt sind, dann legen diese die Operationsreihenfolge fest.

Compute unterstützt Atomvariablen. Wenn alle Atomvariablen mit einem numerischen Typ belegt sind, ergibt Compute einen numerischen Wert als Ergebnis. Nicht numerische Werte ergeben für das Compute als Ergebniswert NIL. Compute verwendet bei der Berechnung Integer, außer wenn einer oder beide der Operanden Double sind, so wird Double verwendet. Ein Double genügt also, damit das Endergebnis einer Compute Funktion Double ist. Compute unterstützt Plus „+“, Minus „-“, Mal „*“, Geteilt „//“ und Modulo „\\“. Negative Vorzeichen einer Atomvariable muss man genauso wie in Lisp OPS5 [Kri87, S. 51] nicht als „- <atomvar1>“ sondern als „0 - <atomvar1>“ schreiben.

Innerhalb von Compute sind keine weiteren Compute Aufrufe erlaubt. [Kri87, S. 49] zeigt, dass Compute schon konkret vor dem Propagieren berechnet wird und dann nur noch dieser konkrete Ergebnis Wert weiter verwendent wird. In MemoRete OPS5 muss Compute aber erst am Ende beim feuern einer Produktionsregel ausgerechnet werden.

4.6.7 Aktion Halt

Aktion Halt sieht so aus: `(halt)`. Wenn sie feuert wird der Recognize-Act-Zyklus des Rete Algorithmus gestoppt. Es werden alle Aktionen der zuletzt gefeuerten Produktionsregel ausgeführt, auch wenn das `(halt)` sich vor anderen Aktionen im Aktionsteil befindet. Vergleiche Methode

```
public boolean fire(BigInteger time) throws Exception {...}
```

in [Fra24, Dateipfad: MemoReteOPS5/src/main/ops5/workingmemory/data/ReteEntityWrapper.java]). Wenn MemoRete OPS5 im interaktiven Modus gestartet wurde, so gelangt der Benutzer nach der Aktion Halt wieder zur interaktiven Kommandozeile. Andernfalls wird das Programm beendet.

4.7 Änderungen in MemoRete OPS5

Es gibt eine Menge an Operationen in Lisp OPS5 und Vax OPS5, die in MemoRete OPS5 nicht unterstützt werden:

Die im Lehrbuch [Kri87, S. 36] beschriebenen `literalize`-Alternativen `Literal` und `Vector-attribute` werden nicht unterstützt, da MemoRete OPS5 ein anderes Arbeisspeicherlayout verwendet. Die Anweisung Literal aus dem Lehrbuch:

```
(LITERAL
  Name = 5
  Vorname = 5
  Nachname = 6)
(MAKE Person ^Name Konrad Zuse)
```

würde es beispielsweiße ermöglichen die Indizes der Attribute explizit festzulegen. Außerdem wird die Belegung aufeinanderfolgender Indizes wie im Beispiel nicht unterstützt.

Substring wird nicht unterstützt z.B. `(SUBSTR <Person> 2 3)`.

`(GENATOM)` wird nicht untersützt. Laut Lehrbuch [Kri87, S. 53] kann man es benutzen um eine fortlaufende individuelle ID als Objektattribut zu erstellen. Man kann das aber auch mit anderen Programmkonstrukten lösen.

`(LITVAL)` wird nicht unterstützt, da es mit der Indizierung der Attribute zu tun hat.

Im Lehrbuch [Kri87, S. 57] steht, dass wenn in OPS5 zwei Modify Aktionen auf die gleiche Elementvariable (z.B. `<Person>`) angewendet werden, dann nur eine gelöscht wird, aber zwei erstellt werden. Das ist sehr unerwartet für einen Benutzer. Man würde wahrscheinlich erwarten, dass die Person `<Person>` zweimal modifiziert wird, und nicht eine zweite erstellt. Um dieses Problem zu lösen ist es in MemoRete OPS5 nicht möglich die gleiche Elementvariable öfter als einmal zu verwenden. Sichergestellt wird das im Konstruktor für Produktionsregeln in [Fra24, Dateipfad: MemoReteOPS5/src/main/ops5/workingmemory/data/ProductionRule.java].

`(BIND)` ist in seiner Funktion redundant mit `(MODIFY)` und wird deshalb nicht implementiert.

`(CBIND)` „bindet das letzte in den Arbeitsspeicher geschriebene Element an eine Elementvariable“ [Kri87, S. 58]. Dieses Sprachkonstrukt wird in MemoRete OPS5 nicht gebraucht. Was mit `(CBIND)` gemacht wird, kann auch schon mit dem davor stehenden `(make ...)` oder `(modify ...)`, auf das sich das `(CBIND)` bezieht, gemacht werden. Lediglich bei der Aktion Call würde sich

(CBIND) lohnen, dem wird aber schon Abhilfe geschafft durch die Subaktionen Make, Modify und Remove.

(BUILD ...) ermöglicht es Produktionsregeln zur Laufzeit zu erstellen (vgl. [Kri87, S. 59]). Das ermöglicht selbstmodifizierenden Code, wird aber in MemoRete OPS5 nicht unterstützt. Bei Bedarf könnte man diese Funktionalität noch implementieren.

Bedingungs-Funktionen nach VAX-OPS5 Art werden nicht unterstützt (vgl. [Kri87, S. 39]) z.B.: (Kiste ^Farbe {<Farb_Variable> <> rot})

```
; MemoRete OPS5 example program

(strategy LEX)

(literalize robot
        name
        type
        velocity
        location
        instantiated
)

(literalize pose
        name
        x
        y
)

(literalize init val)

(literalize cleanup)

(startup
    (make robot ^name SIM_DX1 ^type Pioneer-AT ^velocity 0 ^location pat1)
)

(p makeInitial
    -(init)
    (robot)
-->
    (make init ^val 1)
)

(p moveRobot1to2
    {<elemvariable_init> (init ^val 1 ^val <value>)}
    {<robo> (robot ^name <na> ^location <locationName> ^velocity <velo>)}
    (pose ^name <locationName>)
-->
    (call <na> emo.ifs.ecar.ECar.speed ^transVel 200)
    (call <na> emo.ifs.ecar.ECar.rotate ^transVel 30)
    (modify <robo>  ^velocity 200)
    (modify <elemvariable_init> ^val (compute 1+<value>))
)

(p wait2to3
    {<elemvariable_init> (init ^val 2 ^val <value>)}
    (robot ^name <na> ^location <locationName>)
    {<pose> (pose ^name <locationName>)}
-->
    (call <na> emo.ifs.ecar.ECar.wait ^cycles 100)
    ;wait will also stall rete execution.
    (modify <elemvariable_init> ^val (compute 1+<value>))
    (remove <pose>)
)

(p detectfinished
    {<elemvariable_init> (init ^val > 2)}
-->
    (make cleanup)
```

```
60 )
61
62 (p cleanup1
63     (cleanup)
64     {<robo>(robot ^name <na>)}
65 -->
66     (call <na> emo.ifs.ecar.ECar.disconnect)
67     (remove <robo>)
68 )
69
70 (p cleanup2
71     (cleanup)
72     -(robot)
73 -->
74     (write |Cleanup done. Exit program.| (CRLF))
75     (halt)
76 )
77
78 (p printRobo
79     (robot ^name <na>
80         ^type Pioneer-AT
81         ^type <type>
82         ^velocity <velo>
83         ^location <locationName>
84         ^instantiated true)
85     (pose ^name <locationName>
86         ^x <x> ^y <y>)
87 -->
88     (write |robot | <na>
89         | of type | <type>
90         | speed | <velo>
91         | and location (| <x>
92         |, | <y>
93         |)| (CRLF)
94     )
95 )
96
97 (p instantiateAndConnectRobo
98     {<robo>(robot ^name <na> ^instantiated NIL)}
99 --> (call <na> emo.ifs.ecar.ECar.new ^eCarName <na> ^world JWorld
100         => modify <robo> ^instantiated true)
101     (call <na> emo.ifs.ecar.ECar.connect)
102 )
103
104 (p storeRoboLocationAndSpeed
105     (robot ^name <na> ^instantiated true ^location <location>)
106     -(pose ^name <location>)
107 --> (call <na> emo.ifs.ecar.ECar.getPosition
108         => make pose ^name <location> ^x <?>)
109 )
110
111
112
113
114
115
```

Abbildung 4.1: MemoRete OPS5 Befehlssatz Beispiel

Kapitel 5

Rete Algorithmus Implementierung

Der Hauptteil des MemoRete OPS5 Programms ist der Rete Algorithmus. Er ist implementiert im Code in [Fra24, Dateipfad: MemoReteOPS5/src/main/ops5/ModelRete.java]. Der Rete Algorithmus gleicht eine Menge von Produktionsregeln mit einer Menge von Fakten ab. Er nutzt dabei die Eigenschaft, dass sich von einem Recognize-Act-Zyklus zum nächsten die Faktenbasis nur wenig verändert. Abbildung 5.1 zeigt eine Übersicht des Algorithmus.

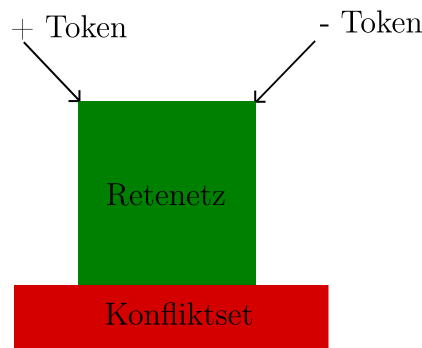

Abbildung 5.1: Recognize-Act-Zyklus

Die Ausführung eines Recogniz-Act-Zyklus ist der Kern des Rete-Algorithmus, er wird durch die Methode `boolean executeStep()` implementiert. Die grüne Box symbolisiert das Retenetz. Es testet Produktionsregeln mit Fakten entsprechend der Recognize Phase. Die rote Box, das Konfliktset, ist eine geordnete Menge aller Produktionsregeln die durch Fakten erfüllt sind. Wenn mindestens eine Regel passt, so wird entsprechend der Act Phase die erste aller passenden gefeuert. In einem Zyklus wird nur maximal eine Regel gefeuert. Erzeugt die gefeuerte Regel einen Fakt, so entsteht ein Plus-Token (vgl. `newAddEntities` in [Fra24, Dateipfad: MemoReteOPS5/src/main/ops5/workingmemory/node/Node.java]). Löscht sie einen Fakt, so entsteht ein Minus-Token (vgl. `newRemoveEntities` in `Node.java`). Die Aktion Modify erzeugt einen Plus und einen Minus Token. Auf diese Art repräsentieren die Token alle möglichen Änderungen der Faktenbasis. Nach Ausführung der Act Phase ist ein Durchlauf des Recognize-Act-Zyklus abgeschlossen.

Zur Ausführung eines MemoRete OPS5 Programms läuft der Recognize-Act Zyklus so lange bis keine Regeln mehr passen, oder die Aktion Halt aufgerufen wurde.

Das **Retenetz** besteht aus jeweiles einem Wurzelknoten pro Literal, aus Alpha-, (existenz und nichtexistenz) Beta- und Terminierungs-Knoten. Alle Knoten befinden sich in [Fra24, Dateipfad: MemoReteOPS5/src/main/ops5/workingmemory/node/].

5.1 Fakt (Fact)

Die Menge aller Fakten ergeben die Faktenbasis. Fakten sind implementiert in [Fra24, Dateipfad: MemoReteOPS5/src/main/ops5/workingmemory/data/Fact.java]. Ein Fakt ist ein Java Record. Damit sind alle im Fakt enthaltenen Daten unveränderlich. Die einzelnen Attribute im Fakt sind nur per Attributnamen adressierbar und nicht per Index. So kann der Speicher für alle Attribute mit `NIL` gespart werden. Wird ein Attribut per Index referenziert, so geht das nur über einen Nachschlag der Attributliste beim entsprechenden Literal.

Im Rete Algorithmus kann die Anzahl der Fakten erheblich ansteigen. Die Anzahl der Regeln ist dagegen sehr gering, da diese vom Nutzer per Hand programmiert werden. Einzig wenn die Anweisung `(build ...)` implementiert wäre, könnte es mehr Regeln als Fakten geben. Da das aber nicht der Fall ist, ist für den Speicherverbrauch die größe der Fakten wichtiger.

Das Lehrbuch [Kri87, S. 32] erwähnt einige Limitierungen für Fakten, so ist die „maximale Anzahl von Objekten [...] in VAX-OPS5 auf 8191 begrenzt.“ Da in MemoRete OPS5 `LinkedHashMultiset<ReteEntity>` und ähnliche Datentypen verwendet werden, liegen die Limits bei mehr als 2^{30} für die HashMap Leistung. Danach kann es sein dass die Leistung etwas einbricht, die Programmfunktion ist aber gewährleistet. Einzig wenn die Anzahl gleicher Fakten `Integer.MAX_VALUE`, also $2^{31} - 1$, überschreitet könnnen diese in den verwendeten `Multiset` nicht mehr gespeichert werden und es gibt einen Fehler.

Die verwendeten Java Records implementieren implizit die Methoden `hashCode()` und `equal()`. Zwei gleiche Fakten besitzen damit den gleichen Hashcode. Für die eigentliche Propagierung über die Knoten des Retenetzes werden Fakten durch Rete Einträge vertreten.

5.2 Rete Eintrag (ReteEntity)

Rete Einträge sind implementiert durch den Record ReteEntity in [Fra24, Dateipfad: MemoReteOPS5/src/main/ops5/workingmemory/data/ReteEntity.java]. Rete Einträge befinden sich auf allen Knoten und repräsentieren den momentanen Stand der Propagierung von Fakten durch das Retenetz. Ein Rete Eintrag enthält entweder einen Verweis auf einen Fakt (wenn es sich auf einem Wurzel- oder Alpha-Knoten befindet) oder einen Verweis auf ein Beta Tupel (wenn er sich auf einem Beta Knoten oder Terminierungs-Knoten befindet). Auf diese Art bilden die Rete Einträge eine Baumstruktur ausgehend von einer oder mehreren Wurzel-Knoten hin zu einem Terminierungs-Knoten wenn die Propagierung nicht schon vorher stoppt. Durch diese Baumstruktur kann später beim feuern der Regeln von einem Eintrag im Terminierungs-Knoten über die Referenzen Knoten für Knoten zurück auf die Ausgangsfakten geschlossen werden. Andersherum kann von der Wurzel durch neu testen oder Speicherung im Knoten auf die darauffolgenden Rete Einträge geschlossen werden.

Diese Art der Rete Einträge benötigt nicht wie im herkömmlichen Rete-Algorithmus das kopieren des ganzen vorhergehenden Fakts oder Rete Eintrags, sondern es benötigt nur entweder eine Referenz (bei Alpha-Knoten oder bei Beta-Knoten mit nur einem vorausgehendem Knoten) oder zwei Referenzen (bei Beta Knoten mit zwei vorausgehenden Knoten). So ist auch bei Produktionsregeln mit vielen Bedingungen der Speicherverbrauch pro Teilmatch konstant anstatt linear wachsend mit der Anzahl an Bedingungen. Bezahlt wird das durch einen größeren Aufwand beim Auflösen von Atomvariablen, da der Wert nun über mehrere Indirektionen abgerufen werden muss.

Da Rete Einträge in Knoten (vgl. `Node.java`) in `LinkedHashMultiset<ReteEntity>` gespeichert werden ist die Leistung der `hashCode()` Methode von `ReteEntity.java` wichtig. Relevant für den Hashcode und den Gleichheitsvergleich ist nur `Object entity`. Würde man den Hashcode bei jedem Aufruf neu berechnen, so müsste der ganze obere Baum an Rete Einträgen den Hashcode neu ermitteln. Um das zu beschleunigen wird der Hashcode fest im Rete Eintrag abgespeichert und ist somit O(1).

Das hinzufügen und entfernen von Rete Einträgen im Retenetz ist im originalen OPS5 rekursiv. In MemoRete OPS5 ist es statt dessen jeweils pro Knoten implementiert. Ein Knoten bekommt ein Multiset `ImmutableMultiset<ReteEntity>` an Rete Einträgen zur Verarbeitung auf einmal. Durch diese schubweise Abarbeitung kann der CPU Cache besser genutzt werden.

Des Weiteren ermöglicht das verwenden von Multisets einen Speicherverbrauch von $O(1)$ für das speichern gleicher Rete Einträge. Die Methoden in `Node.java` zur Propagierung

```
public abstract void propagateAddedEntities(ImmutableMultiset<
    ReteEntity> reteEntities, Node from, BigInteger time);
public abstract void propagateRemovedEntities(ImmutableMultiset<
    ReteEntity> reteEntities, Node from, BigInteger time);
```

würden es auch erlauben mehrere gleiche Rete Einträge auf einmal zu verarbeiten und so die Propagierung von n gleichen Einträgen `reteEntities` von $O(n)$ auf $O(1)$ zu senken. Diese Eigenschaft wurde aber aufgrund der Komplexität (speziell in den Beta-Knoten) noch nicht implementiert. In einer zukünftigen Programmversion ist es aber möglich und durchaus wünschenswert.

Überlegt wurde auch die Rete Einträge „träge“ zu propagieren. Damit würde man im Schnitt weniger Rechenleistung für ein Match benötigen. Deren Realisierbarkeit hängt aber von der Strategie `strategy`, und Beta-Knoten die auf Nichtexistenz prüfen, ab. `strategy` würde definieren welche Rete Einträge und Knoten als erstes propagiert werden müssten. Weiter müssten Beta-Knoten die auf Nichtexistenz prüfen, wenn sie nach der `strategy` dran kommen, alle Teilmatches auf Nichtexistenz geprüft haben müssen. So müsste der Algorithmus „träge“ nach den Vorgaben der `strategy` Knoten und Rete Einträge prüfen. Eine solche „träge“ Propagierung wurde nicht entwickelt, sie hätte aber bessere Laufzeiteigenschaften da sie nicht alle Teilmatches bilden muss. Es wurde nicht genauer darauf eingegangen ob die Strategien `LEX` und `MEA` wirklich eine solche Propagierung erlauben.

Das Löschen eines Rete Eintrags erfordert rekursives nach oben gehen zu den Wurzel-Knoten. Dort wird der Eintrag im nächsten Recognize-Act-Zyklus gelöscht. Die Löschung propagiert bei diesem neuen Zyklus dann nach unten und entfernt alle zugehörigen Teilmatches bis einschließlich zu dem Match im Terminierungs-Knoten, der gefeuert hat. Auf dem Weg müssen bei den betroffenen Knoten die Rete Einträge mit `HashSet.contains()` gefunden und gelöscht werden. Das ist jeweils $O(1)$. Gefunden wird der Fakt durch die Methode „`public Fact getElemvarFact(String elemVar)`“. Sie kann rekursiv oder als Schleife implementiert werden. Im Code wurde sie als Schleife implementiert, da dies schneller ist und nicht die Gefahr eines Stapelüberlaufs hat. Es verfolgt die Rete Einträge nach oben immer mit der Wahl der richtigen Verzweigung an den Beta-Knoten. Ähnlich ist es mit der Methode „`public <T> Value <T> getValue(String atomVar)`“ die für Atomvariablen benutzt wird. Würde Java Tail-Call Optimierung unterstützen, so könnte stattdessen ein rekursiver Ansatz verwendet werden.

=	gleicher Typ und equal
<>	anderer Typ oder unequal
<=>	gleicher Typ
<	kleiner
<=	kleiner gleich
>	größer
>=	größer gleich

Abbildung 5.2: Vergleichsoperationen

5.3 Wert (Value)

Attributwerte von Fakten werden in Wert gespeichert (vgl. [Fra24, Dateipfad: MemoReteOPS5/src/main/ops5/workingmemory/data/Value.java]). Ein Wert kann einen `String`, ein `Double`, ein `Integer` oder `NIL` speichern. Javas `null` entspricht einem `NIL` Wert um Speicher zu sparen.

Wert ermöglicht das vergleichen mit anderen Werten. Es gibt folgende Vergleichsoperationen:

Größenvergleiche funktionieren nur zwischen numerischen Typen. Wird mit einem nichtnumerischen Typ verglichen, so wird die Wertekombination nicht propagiert. Im Gegensatz zu Lisp OPS5 unterstützen numerische Typen in MemoRete OPS5 auch Größenvergleiche zwischen `Integer` und `Double`.

Man hätte auch Vergleiche zwischen Strings nach alphabetischer Reihenfolge zulassen können, das würde aber Probleme bei Vergleichen zwischen Strings und numerischen Werten geben.

5.4 Knoten, Propagierung und Strategie

Knoten, die Propagierung von Rete Einträgen und die Regel-Feuer Strategien hängen stark zusammen. Diese Sektion behandelt diese und deren Zusammenhänge.

Knoten sind in [Fra24, Dateipfad: MemoReteOPS5/src/main/ops5/workingmemory/node/] implementiert. Sie werden zusammen gekettet zum Retenetz. Redundante Knotenlisten werden nur einzigartig erstellt um Speicher und Rechenzeit zu sparen (siehe Methode `addRules(Parser parser)`). Über die Knoten vom Retenetz, von den Wurzel- hin zu den Terminierungs-Knoten propagieren die Rete Einträge. Das ist im Code in [Fra24, Dateipfad: MemoReteOPS5/src/main/ops5/ModelRete.java] in der Methode `propagateEntities()` umgesetzt.

Die Methode `propagateEntities()` nutzt eine Warteschlange

```
final private UniqueQueue<Node> nodesWithChangedEntities
```

in der Knoten zur Propagierung nur einzigartig hinzugefügt werden können. Wann immer ein Knoten neue Ergebnis Rete Einträge in `processedEntities` generiert oder löscht, wird der Knoten in die Warteschlange zur Propagierung eingereiht. Da Einträge in die Warteschlange immer am Ende sind kann es sein dass der gleiche Knoten viele Eintragsänderungen bekommt bis er an er Spitze der Warteschlange angelangt ist. So werden in den Knoten mehr Eintragsänderungen auf einmal verarbeitet, was den Prozessorcache besser nutzt.

Die Strategie `strategy` definiert die Reihenfolge zum feuernden der Regeln und Matches (Matches im Konfliktset sind verpackt in `ReteEntityWrapper`). Alle potenziellen Matches befinden sich im Konfliktset [Fra24, Dateipfad: MemoReteOPS5/src/main/ops5/workingmemory/data/ConflictSet.java]. Die Strategien `LEX` und `MEA` bestimmen die Reihenfolge der Einträge im Konfliktset. Alle Matches die feuern können sind in `conflictSet`. Wenn sie gefeuert haben werden sie nach `conflictSetFired` verschoben. Wenn das Retenetz ein Match löscht, so wird das entsprechende Match erst versucht im `conflictSetFired` zu löschen, ansonsten wird es aus dem `conflictSet` gelöscht.

Die Rete Einträge im `conflictSet` sind in einem `TreeMultiset<ReteEntityWrapper>` gespeichert, welches als Sortierung `makeComparator(Strategy strategy)` entweder von `Strategy.LEX` oder von `Strategy.MEA` benützt. Der verwendete `Comparator` bringt die Matches in eine totale Ordnung durch Verwendung einer `ComparisonChain`. Die zum Ordnen nötigen Daten `creationTime` und `creationTimeMostLeft` befinden sich im darunter liegenden Rete Eintrag und der `nodeBetaSpecificityScore` sowie der `nodeAlphaSpecificityScore` befinden sich auf den Terminierungs-Knoten.

Ein wesentlicher Nachteil für den Speicherverbrauch ist, dass jeder Rete Eintrag die zwei zusätzlichen Zeitstempel speichern muss. Dies ist aber notwendig für die Strategien. Weiter ist das Ergebnis von `equal()` ungleich dem vom `Comparator`. Equal darf nämlich nur die gespeicherten Fakten oder Beta Tupel Vergleichen und die Zeitstempel nicht berücksichtigen. Sonst wäre das Propagieren durchs Retenetzwerk von gleichen Rete Einträgen falsch. Damit man aber Matches aus dem `conflictSet` löschen kann, benötigt man den Rete Eintrag mit dem richtigen Zeitstempel. Die Zuordnung der `equal()` zu den `Comparator` Matches befindt sich in:

```
private HashMap<ReteEntityWrapper, TreeMultiset<ReteEntityWrapper
  >> conflictSetTracker;
```

Durch diese Zuordnung kann für jedes gelöschte Match dasjenige Match mit den richtigen Zeitstempeln gefunden werden, dass dann aus `conflictSet` gelöscht werden kann.

Da die Propagierung durch das Netzwerk mit einer Warteschlange funktioniert könnte man es in den einzelnen Knoten mit Mutexen isolieren um parallel mit mehreren Prozessthreads die Warteschlange abzuarbeiten. Das ist nicht implementiert.

Da im Konfliktset die Matches entweder nach einem festen Schema oder Zufällig geordnet sein müssen, kennt man immer das Prinzip das entscheidet welche Regel als erstes feuert (Zufälligkeit ist auch ein Prinzip). Da man also das Regelfeuerprinzip kennt kann man jene Fakten und Teilmatches priorisiert propagieren, die nach dem Regelfeuerprinzip als erstes feuern. Auf diese Art könnte man das erstellen vieler Teilmatches überspringen. Es wurde aufgrund der Komplexität nicht implementiert. Ein möglicher Stolperstein bei einer solchen Implementierung sind die Beta-Knoten die auf Nichtexistenz testen.

5.5 Alpha-Knoten

Für jede Bedingung einer Produktionsregel wird eine Liste von aufeinanderfolgenden Alpha-Knoten erstellt. Existiert schon eine ähnlich Folge von Alpha-Knoten, so werden diese vom ersten Alpha-Knoten an so weit wie möglich wiederverwendet. Ein Alpha-Knoten implementiert eine Teil-Bedingung `ConditionFilter` (vgl. [Fra24, Dateipfad: MemoReteOPS5/src/main/

ops5/workingmemory/data/condition/]) und propagiert nur die Rete Einträge, die die Bedingung erfüllen.

Kondition „`ConditionFilter filter`“ filtert alle Rete Einträge die durch den Alpha-Knoten gehen. Für Konditionen mit Atomvariablentests wird die Belegung der Atomvariable mit der Methode `getValueRestriction` ermittelt:

```
public <T> ValueRestriction<T> getValueRestriction(String
    atomVarName, ReteEntity reteEntity)
```

Man könnte Speicherplatz sparen, indem die Rete Einträge in den Alpha-Knoten nur vor einem Übergang zu einem Beta-Knoten gespeichert werden und nicht in den Zwischen-Alpha-Knoten. Das Hinzufügen einer Regel würde dann mehr Aufwand bedeuten. Die Rete Einträge in den betroffenen Alpha-Knoten müssten neu erstellt werden, wenn ein Zwischen-Alpha-Knoten als Quelle für einen Beta-Knoten benötigt wird. Das wurde nicht implementiert, wäre aber möglich.

Bei den Alpha-Knoten kann Filteraufwand gespart werden indem Alpha-Knoten die einen Größenvergleich auf das gleiche Attribut ausführen hintereinander kettet. Beispielsweise beim Vergleich von „`> 4`“ mit „`> 7`“. „`> 4`“ kann also Vorfilter sein für „`> 7`“. So müssen alle Rete Einträge die für den Attributwert größer als 7 sind nicht mehr auf „`> 4`“ getestet werden. Ist in diesem Fall aber ein Wert kleiner gleich 7, so spart man keinen Aufwand und der Test mit „`> 4`“ findet wie sonst auch statt. Das wurde nicht implementiert wäre aber gerade bei mehreren Größenvergleichen auf das gleiche Attribut schneller.

[GR98, S.482] zeigt das Wiederverwenden von Atomvariablen in Alpha-Knoten an einem Beispiel ähnlich:

```
1 Simpel
2 (data ^a <x> ^b <x>)
3 (data ^a <y> ^b <y>)
4 Komplex
5 (data ^a <x> ^b <x>)
6 (data ^b <y> ^a <y>)
```

Weiter beschreibt [GR98, S.482], dass die Atomvariablen `<x>` und `<y>` im simplen Beispiel oben zusammengeführt werden können, jedoch nicht im komplexen. Es ist denkbar, dass ein optimierter Erstellungsalgorithmus für Retenetze auch den komplexen Fall mit dem simplen Fall zusammenführen kann und so für die vier Bedingungen insgesamt nur 2 Alpha-Knoten benötigen würde. Für den simplen Fall benötigt man Atomvariablentransparenz. Für den komplexen Fall benötigt man Invarianz der Alpha-Knoten Reihenfolge, was durch Neusortierung der Alpha-Knoten möglich wäre. Durch letztere Optimierung könnten aber andere überlappende Ketten von Alpha-Knoten aufgebrochen werden. Beides wurde aufgrund der Komplexität nicht implementiert.

In MemoRete OPS5 werden Alpha Knoten unterschieden ob sie auf Existenz oder Nichtexistenz testen. Weil diese Tests final erst bei dem zugehörigen Nichtexistenz-Beta-Knoten entschieden werden können, müssen diese Alpha-Knoten gleich filtern wie Alpha-Knoten die auf Existenz testen. Das ermöglicht es beide, Existenz und Nichtexistenz, Alpha Knoten zusammen zu legen. Das wurde nicht implementiert, könnte aber redundante Alpha-Knoten sparen.

5.6 Beta-Knoten

Die Beta-Knoten sind das Herz und der komplexeste Teil von MemoRete OPS5 (befindlich im Code bei den anderen Knoten). Man unterscheidet Existenz-Beta-Knoten und Nichtexistenz-Beta-Knoten. Gemein ist beiden aber die abstrakte Klasse `NodeBeta.java`. Beta-Knoten haben zwei oder einen vorausgehenden Knoten. Auf der linken Seite ist potenziell ein anderer Beta-Knoten und auf der rechten Seite ist immer ein Alpha-Knoten. Der Alpha Knoten ist das Ende einer Kette von Alpha-Knoten welche zusammen genau eine Bedingung des Bedingungsteils repräsentieren. Dieses Schema ist ähnlich dem der Regelengine Jess. Es erleichtert die Implementierung von Beta-Knoten, da so die Position schon den Knotentyp festlegt.

Beta-Knoten speichern welche Atomvariable und welche Elementvariable vom linken oder vom rechten Vorgängerknoten kommt. So kann später beim feuern der Regel der richtige Attributwert bzw. der richtige Fakt gefunden werden. Außerdem speichert es auch die Schnittmenge an Atomvariablen vom rechten und linken Knoten in `atomVarsIntersection`. Jede Atomvariable dieser Schnittmenge bedeutet, dass ein Vergleich von den Rete Einträgen der rechten und linken Seite stattfinden muss. Des Weiteren merkt es sich in der Variable `atomVarToPredicate` welche Atomvariable mit welchem `ConditionPredicate` verglichen wird, also größer, kleiner, gleich, usw.

In der Methode `setVarRefs(...)` werden diese Variablenbeziehungen gebildet. Besonders bei Atomvariablen muss geprüft werden, wenn sie auf beiden Seiten vorkommen, ob der linke Beta-Knoten eine equal „=“ Zuweisung hat. Das ist gleich wie im original OPS5 und notwendig um die Vergleiche durchführen zu können. Wenn das nicht der Fall wäre, so könnte man aufeinanderfolgende Bedingungen wie `<atomVar1> > 7` und `<atomVar1> <> 10` programmieren. Da dort aber nie die Atomvariable `<atomVar1>` einen tatsächlichen Wert zugewiesen bekommt wäre es für den Rete Algorithmus nicht entscheidbar ob einerseits die Bedingung zutrifft und andererseits welchen Wert die Atomvariable im Aktionsteil haben soll. Nachdem aber die Atomvariable einen Wert zugewiesen hat, können beliebig viele Bedingungen auf sie folgen, die keinen equal „=“ Vergleich durchführen, also größer, kleiner, gleicher Typ, etc.

Man könnte die Zuweisung einer Atomvariable auch später zulassen. So würden die einzelnen Bedingungen immer nur jeweils den Wertebereich oder den Typ einschränken. Das ist aber nicht möglich, da spätestens bei einem Test auf ungleichheit ein konkreter Wert vorliegen müsste. Außerdem muss vor einem Terminierungs-Knoten immer mindestens eine konkrete Wertzuweisung sein, du nur so die Atomvariable im Aktionsteil benutzt werden kann.

Man könnte beim erstellen des Retenetzes die Beta-Knoten in eine intelligente Reihenfolge bringen, sodass Produktionsregeln eine minimale Anzahl an Beta-Knoten erzeugen. Das wäre möglich, wurde aber nicht implementiert. Will man viele Überlappungen erzeugen, so muss der Benutzer das OPS5 Programm so erstellen, dass die Bedingungsteile von Produktionsregeln möglichst ähnlich beginnen. Das Ändern der Reihenfolge der Bedingungen, als auch der Bedingungsteile kann dabei helfen.

Alternativ zu den Beta-Knoten mit nur zwei Vorgänger-Knoten könnte man auch Beta-Knoten erzeugen die n vorausgehende Knoten haben. Wenn es z.B. keinen nutzen hat 2er Beta-Knoten zu erzeugen, da deren Teilmatches nur von einem nachfolgenden Knoten genutzt werden, so könnte man diese zwei 2er Beta-Knoten zu einem 3er Beta-Knoten zusammenfügen. Das kann man so weiterführen bis zu beliebigen n-Betaknoten. Besonders wenn sich das Retenetz nach dem initialen Aufbau nicht mehr ändert wäre das schneller. Die Teilmatches aller

Zwischenknoten würden wegfallen und man könnte z.B. wenn ein Zwischenknoten keine Rete Einträge durch seinen Alpha-Knoten einbringt einfach keine Ergebnis Rete-Einträge erzeugen. Das wurde nicht implementiert, wäre aber eine sehr gute Lösung. Nur die Aktion `(build ...)` zur dynamischen Produktionsregel-Erzeugung könnte die Vorteile verringern, da es dann vorkommen könnte dass n-Beta-Knoten zur Laufzeit aufgebrochen werden müssen. Bei spärlicher Nutzung von `(build ...)` würden aber immer noch die erheblichen Vorteile in Laufzeit und Speicherverbrauch überwiegen.

In den Beta-Knoten von MemoRete OPS5 werden die Ergebnis Rete Einträge auf den erzeugenden Beta-Knoten selbst gespeichert in `processedEntities`. Man könnte die Ergebnisse auch auf allen folgenden Knoten speichern, das würde aber den Speicherverbrauch erhöhen.

5.6.1 Existenz-Beta-Knoten ohne Atomvariable

Bei den Existenz-Beta-Knoten werden alle Rete Einträge von der linken mit allen Rete Einträgen von der rechten Seite gematcht. Wenn die Schnittmenge an Atomvariablen leer ist, also `atomVarsIntersection` keine Elemente hat, dann werden alle Rete Einträge mit allen Rete Einträgen der anderen Seite erfolgreich gematcht. Wenn also die erste Bedingung n Rete Einträge und die zweite m Rete Einträge liefert, so entstehen in dem Beta-Knoten $n*m$ Ergebnis Rete Einträge. Eine Liste von Existenz-Beta-Knoten produziert nach dem gleichen Schema $n*m*o*p*...$ Ergebnis Rete Einträge. Nennt man k die Menge an Bedingungen die auf Existenz prüfen, und n die Menge an Fakten die jede Bedingung erfüllen, so werden n^k Ergebnis Rete Einträge erzeugt. Ist in dem Existenz-Beta-Knoten keine Atomvariable in der Schnittmenge von Atomvariablen, so ist das auch das Verhalten von MemoRete OPS5.

5.6.2 Existenz-Beta-Knoten mit Atomvariable

Beim Vorhandensein von Atomvariablen in der Schnittmenge werden nur jene Kombinationen Teil der Ergebnis Rete Einträge, welche alle Bedingungen „`<,<=,>,>=`“ erfüllen. Im klassischen Rete Algorithmus müssen dazu aber immer noch alle Kombinationen getestet werden. In MemoRete OPS5 ist das verbessert. Näheres dazu ist in Sektion Multiset und den folgenden.

5.6.3 Nichtexistenz-Beta-Knoten ohne Atomvariable

In Nichtexistenz-Beta-Knoten nehmen wir zuerst wieder an, dass die Schnittmenge an Atomvariablen leer ist.

In Nichtexistenz-Beta-Knoten lassen sich Rete Eintrags Änderungen und deren folgen durch einen endlichen Automaten darstellen mit den Zuständen `(0,0)`, `(0,>0)`, `(>0,0)` und `(>0,>0)` und deren Zustandsübergängen.

Folgende Tabelle 5.3 zeigt die möglichen Zustände eines Nichtexistenz-Beta-Knotens. „`( a)`“ symbolisiert Rete Einträge vom linken Beta-Knoten. „`-(b)`“ symbolisiert Rete Einträge vom rechten Nichtexistenz-Alpha-Knoten. Die Zeile „Propagation“ zeigt die Änderungen der Ergebnis Rete Einträge. Plus $\oplus$ bedeutet das dazukommen und Minus $\ominus$ das löschen eines Rete Eintrags. 0 heißt dass zu dem Moment keine Rete Einträge vorhanden sind. n heißt, dass mindestens ein Rete Eintrag vorhanden ist. Ein leeres Feld bedeutet dass keine Änderung der Rete Einträge stattfindet. Die `if` Bedingungen beziehen sich auf die Anzahl der Rete Einträge von entweder „`(a)`“ oder „`(b)`“.

(a)	0	0	n	n	$\oplus$	$\ominus$	$\oplus$	$\ominus$
-(b)	$\oplus$	$\ominus$	$\oplus$	$\ominus$	0	0	n	n
Propagation			if(b=1)$\ominus$	if(b=0)$\oplus$	if(a=1)$\oplus$	if(a=0)$\ominus$		

Abbildung 5.3: Zustände des Nichtexistenz-Beta-Knoten

Zusammengefasst gilt: Wenn `(b)` keine Einträge hat sind alle Rete Einträge von `(a)` propagiert. Wenn `(b)` mindestens einen Eintrag hat sind keine Rete Einträge von `(a)` propagiert. Dieses „alles oder nichts“ propagieren ist differenzierter bei Nichtexistenz-Beta-Knoten mit Atomvariable.

5.6.4 Nichtexistenz-Beta-Knoten mit Atomvariable

Im Nichtexistenz-Beta-Knoten wird jetzt angenommen, dass die Schnittmenge an Atomvariablen mindestens eine Atomvariable enthält. Eine konkrete Belegung der Atomvariablen in der Bedingung `(a)` kann für eine oder mehrere Rete Einträge in `(a)` gelten. Wenn es nun matches in `(b)` für diese konkrete Belegung von Atomvariablen gibt, so werden keine Ergebnis Rete Einträge propagiert. Gibt es dagegen keine matches in `(b)`, so werden alle Rete Einträge mit der konkreten Belegung von Atomvariablen propagiert.

5.6.5 Multiset

Memorete OPS5 verwendet Multisets zum speichern der Rete Einträge. Multisets sind wie normale Java Sets, nur mit einem zusätzlichen Zähler für jedes Element. Die Methoden `equal (...)` und `hashCode()` von Rete Einträgen berücksichtigen dabei nur das `Objekt entity` im Eintrag, nicht aber die Zeitstempel. So zählen für Gleichheit nur die tatsächlichen Werte. Wenn ein neuer Rete Eintrag zum Beta-Knoten propagiert wird, so muss dieser neue Wert nur mit jedem einzigartigem Rete Eintrag der andere Position verglichen werden. Das spart Rechenaufwand. Große Vergleiche mit vielen gleichen Rete Einträgen müssen dann nur jeweils die einzigartigen Einträge vergleichen.

5.6.6 Multiset pro Wertetupel, Memoisation

Eine weitere Änderung in MemoRete ist die Aufteilung der Rete Einträge in jeweils ein Multiset pro Wertetupel. Ein Wertetupel (im Code auch ValueList genannt) ist eine Liste von Werten `ImmutableList<Value<?>`. Die Wertetupel enthalten einen Wert für jede Atomvariable der Schnittmenge des Beta-Knotens. Wenn die Schnittmenge leer ist, so gibt es nur ein Wertetupel mit länge null. Diesem Wertetupel sind dann alle Rete Einträge zugeordnet. Gibt es mindestens eine Atomvariable in der Schnittmenge, so gibt es ein Wertetupel für jede einzigartige Belegung der Atomvariablen. Diese einzigartige Belegung enthält dann bei sich ein Multiset an Rete Einträgen welche alle diese Belegung der Atomvariablen erfüllen. Die Zuordnung von Rete Einträgen zu Wertetupeln ist relevant für das weitere Funktionieren von MemoRete. Sie ist implementiert in [Fra24, Dateipfad: MemoReteOPS5/src/main/ops5/workingmemory/data/Memoization.java]). Die Wertetupel sind sind als Idee zur Optimierung aus dem von Matching in Nichtexistenz-Beta-Knoten mit Atomvariable hervorgegangen, sind aber generell auf Beta-Knoten anwendbar.

Weiter ist korrektes Speichermanagement in `Memoization.java` notwendig. In einer früheren Phase das Programms, wenn ein Wertetupel keine Zugeordneten Rete Einträge mehr besaß,

wurde das Wertetupel selbst nicht gelöscht. Dieses Speicherleck ist jetzt aber in der Methode `remove(...)` repariert.

Die Klasse heißt Memoization, da sie eben dieses Prinzip (Memoisation) verwendet. Es ermöglicht, dass das testen von Wertetupeln in Beta-Knoten nur beim ersten Rete Eintrag eines Wertetupels notwendig wird, das Ergebnis des Matches also nur einmal zu beginn ermittelt werden muss. Es verringert so die kombinatorische Last indem man anstatt jeden linken mit jedem rechten Rete Eintrag testen zu müssen, man stattdessen nur deren einzigartige Wertetupel testen muss. Diese Vorgehensweise „ist ähnlich einer Materialized view“ (übersetzt aus [Wik06a]) in Datenbanken.

5.6.7 Viele-zu-viele Memoisation

Folgende Abbildung 5.4 zeigt eine Übersicht der Speicherelemente des Algorithmus. Man kann die Grafik in 4 Spalten einteilen, welche jeweils mit den Nachbarn verbunden sind. Die Erste mit der Zweiten, sowie die Vierte mit der Dritten sind die Memoisationen von Rete Einträgen zu Wertetupeln. In der Mitte von der zweiten zur dritten Spalte sind Wertetupel mit matchenden Wertetupeln verbunden.

Wenn ein neuer Rete Eintrag dazukommt der die gleiche Atomvariablenbelegung hat, wie ein bestehendes Wertetupel, so kann er einfach zu den anderen Rete Einträgen mit gleichen Wertetupel hinzugefügt werden. Um dann zu ermitteln mit welchen Rete Einträgen der neue Eintrag matcht müssen nur die Linien (Referenzen) zu den Einträgen auf der jeweils anderen Seite verfolgt werden. Dieser Fall ist sehr effizient in der Laufzeit, weil die ganze Matching-Phase für den Beta-Knoten wegfällt und man direkt die Ergebins Rete Einträge ermitteln kann. Die gleiche Eigenschaft gilt auch für jedes löschen von Rete Einträgen. Im herkömmlichen Rete Algorithmus müssen dagegen beim löschen alle Matches erneut gebildet werden.

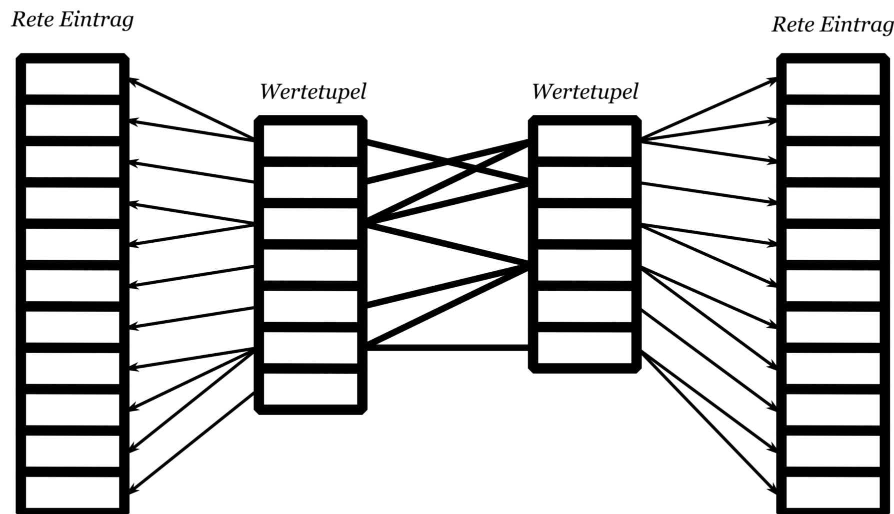

Abbildung 5.4: Speicher im Beta-Knoten

Die viele-zu-viele Verbindungen von Wertetupeln zu Wertetupeln (in der Mitte der Grafik 5.4 und in den Beta-Knoten `reteEntityManyToManyLookup` genannt) werden gespeichert

in der Klasse [Fra24, Dateipfad: MemoReteOPS5/src/main/utils/SpecialBiMultiMap.java]. Es speichert welche Wertetupel mit welchen Wertetupeln gematcht haben, egal ob von der rechten oder der linken Seite. In der Klasse `SpecialBiMultiMap<K, V>` befindet sich für jede Seite eine SetMultimap:

```
    private final SetMultimap<K, V> keyToValues;
    private final SetMultimap<V, K> valueToKeys;
```

Belegt werden diese mit Hashset Schlüsseln und Hashset Werten für $O(1)$ Aufrufe:

```
MultimapBuilder.hashKeys().hashSetValues().build();
```

Wird ein Schlüssel `K` oder ein Wert `V` gelöscht oder hinzugefügt, so muss dies immer auf beiden Seiten geschehen.

Wird ein `K` oder `V` gelöscht, so muss bekannt werden, welche Verbindungen mit dem löschen auch verschwinden. Nur wenn diese bekannt sind, können sie richtig propagiert werden. Erst dann können über die entsprechenden Memoisationen die Rete Einträge gebildet werden, die gelöscht werden müssen. Dies wird ermöglicht indem die Methoden `removeKey(K key)` und `removeValue(V value)` ein `HashSet` zurückgeben welches alle `V` oder `K` enthält zu denen die Verbindung gekappt wurde.

Zusätzlich ist es notwendig für Nichtexistenz-Beta-Knoten zu wissen, wenn beim einfügen eines Werts `V` ein Key `K`, der vorher keine Verbindung hatte, jetzt eine Verbindung bekommen hat. Das entspricht einer konkreten Wertebelegung der linken Bedingung eines Nichtexistenz-Beta-Knotens, der jetzt mindestens einen passenden Rete Eintrag auf der rechten Seite bekommen hat. Diese Änderung bedeutet, dass alle Rete Einträge der linken Seite mit dem Wertetupel (Wertebelegung) ihre zuvor propagierten Rete Einträge jetzt löschen müssen (vgl. Kapitel Nichtexistenz-Beta-Knoten mit Atomvariable). Die Methode `putAllAndReturnDifference(...)` ermöglicht das:

```
public HashSet<K> putAllAndReturnDifference(Set<K> keys, V value)
```

Wenn man in Codeeditor Eclipse diese Methode markiert und im Rechtsklick Menü `Open Call Hierarchy` auswählt, sieht man wo diese Methode im Nichtexistenz-Beta-Knoten verwendet wird.

Mit diesen Informationen sollte es dem Leser jetzt möglich sein nachzuvollziehen, wie im Code die Beta-Knoten hinzugefügte und gelöschte Rete Einträge propagieren. Es wird empfohlen mit dem einfacheren Existenz-Beta-Knoten zu beginnen und danach erst den Nichtexistenz-Beta-Knoten anzusehen. Das eigentliche Matching der Wertetupel wird im folgenden Abschnitt behandelt, jetzt kann aber einfach angenommen werden, dass dort alle Kombinationen von Wertetupeln getestet werden (auch wenn das nicht stimmt). Bei der Propagierung in den Nichtexistenz-Beta-Knoten wird dabei die Methode `addUniqueOccurence(...)` und `removeUniqueOccurence(...)` aufgerufen, wenn ein neues Wertetupel erstellt oder gelöscht wurde. Diese erfordern die komplexere Behandlung da Änderungen in `reteEntityManyToManyLookup` notwendig werden.

5.6.8 Wertetupel Nachschlag Matching

Wie schon gesagt ist das Matching von Wertetupeln mit Wertetupeln nicht ein alles-mit-allen test. Es bieten sich hier Indizierungsmechanismen an die das Nachschlagen beschleunigen. Implementiert ist diese Indizierung im Code in der Klasse [Fra24, Dateipfad: MemoReteOPS5/src/main/ops5/workingmemory/data/ValueListCompare.java].

In einer früheren Version wurden die Rete Einträge direkt zum Vergleich verwendet. Sie waren in `Multiset<ReteEntity>` verpackt. Das hat die Datentypen zum Nachschlagen sehr tief verschachtelt und Fehleranfällig gemacht. Jetzt aber werden in MemoRete OPS5 zum Vergleich nur einzigartige Wertetupel verwendet, was es erlaubt Datentypen der Art `Set<Wertetupel>` zu verwenden, wobei ein Wertetupel eine `ImmutableList<Value<?>>` ist.

Bei der Instantiierung von `ValueListCompare` wird eine Liste an Atomvariablen zusammen mit deren Konditionen in

```
ImmutableMap<String, ConditionPredicate> atomVarToPredicate
```

mitgegeben. `ConditionPredicate` ist dabei eine Kondition aus „`=, <=>, >, >=, <, <=, <>`". Die Reihenfolge der Atomvariablen in `atomVarToPredicate` ist gleich der eines jeden Wertetupels bei späteren Vergleichen (alle Datentypen mit dem Prefix `Immutable` in Java Guava erhalten die Iterationsreihenfolge nach der Instantiierung). Im Konstrukor wird dann für jedes Atomvariablen-Konditions Paar eine Datenstruktur zum schnelleren Nachschlagen gebaut. Diese Zuordnungen werden in `mapTypes` und in den einzelnen Nachschlagstrukturen `reteEntitiesLookup*` gespeichert.

Wird ein Wertetupel über `add(...)` hinzugefügt oder über `remove(...)` gelöscht, so geschieht das auch in allen Nachschlagstrukturen `reteEntitiesLookup*`. Um dann zu testen mit welchen Wertetupeln ein gegebenes Wertetupel matcht wird die Methode `getMatches(...)` aufgerufen. Das getestete Wertetupel entspricht einem neuem Wertetupel auf der anderen Seite des `reteEntityManyToManyLookup`. Die Seite muss auch berücksichtigt werden bei Größenvergleichen „`>, >=, <, <=`", da je nach Seite das Vergleichssymbol umgedreht werden muss. Das wird innerhalb der Methode `getMatchesTree` mit der Variable `reverseTree` gemacht. Für die anderen Vergleiche ist die Seite egal.

Der Wertetupel Nachschlag erfordert einiges an extra Code und Daten. Diese skalieren aber linear mit der Anzahl der Wertetupel, welche immer geringer oder höchstens gleich groß der Anzahl der Rete Einträge ist. Alternativ wäre es auch möglich dieses Matching in einem Datenbanksystem zu machen. Vorgefertigte Indizierung in Datenbanken ist schon sehr ausgereift und so hätte man diese nutzen können. Aufgrund dieser Ähnlichkeit sind die Schnittmengen-Operationen in `ValueListCompare` ähnlich den JOIN-Operationen in Datenbanken.

Matching

Wenn die Methode `getMatches` Aufgerufen wird

```
public HashSet<ImmutableList<Value<?>>> getMatches(ImmutableMap<
   String, Value<?>> valuesMap) ...
```

wird das mitgegebene Wertetupel in `valuesMap` auf passende Wertetupel geprüft.

Das geschieht indem bei jedem Nachschlagewerk `reteEntitiesLookup*` nachgelschlagen wird welche Wertetupel mit dem gegebenen Wertetupel zusammenpassen. Das Nachschlagen ist dabei für alle Nachschlagewerke $O(1)$. Jetzt hat man eine Menge von Mengen von Ergebnis-Wertetupeln, eine Menge pro Nachschlagewerk. Aus all diesen Mengen muss nun die Schnittmenge gebildet werden um jene Wertetupel zu finden, die alle Bedingungen erfüllen, also in jeder Ergebnis-Menge vorkommen. Um diese Schnittmengenbildung zu beschleunigen werden sie der größe nach (beginnend mit dem kleinsten) sortiert in `lookupsList`:

```
Collections.sort(lookupsList, mapComparatorAscending);
```

Für jede Ergebnis-Menge wird dafür die Methode `size()` aufgerufen. Es wurde bedacht darauf gegeben, dass diese $O(1)$ sind, nur bei `TreeLookup.size()` ist die Laufzeit dafür $O(treeSize)$.

Die kleinste Ergebnis-Menge ist nun am Anfang der Liste `lookupsList`. Sie wird, wenn sie kein HashSet ist in ein HashSet umgewandelt. Das ist Notwendig, da die Methode `getMatches(...)` ein `HashSet` zurückgibt. In der Schleife wird nun bis zur letzen Ergebnis-Menge iterativ die Schnittmenge gebildet. Die Wertetupel im `HashSet`, die am Ende übrig sind, sind die Wertetupel die mit dem am Anfang gegebenen Wertetupel matchen.

Test auf Gleichheit

Der Test auf Gleichheit wird durch die Variable `reteEntitiesLookupEqual` durchgeführt:

```
final HashMap<ImmutableList<Value<?>>, HashSet<ImmutableList<
   Value<?>>>> reteEntitiesLookupEqual;
```

Er bildet eine neue Schlüssel-Werteliste auf ein Set von Wertetupel ab. In dem Set befinden sich die eigentlichen Werte zum Vergleich. Die Schlüssel-Werteliste setzt sich zusammen aus je einem Wert der auf Gleichheit getestet wird. Wenn also in einem Beta-Knoten die linke und die rechte Bedingung jeweils zwei Atomvariablen auf Gleichheit matcht, so hat die Schlüssel-Werteliste auch zwei Einträge. Je ein Eintrag für eine Atomvariable die auf Gleichheit testet. Unabhängig davon können zwischen der linken und rechten Bedingung weitere Vergleiche sein. Diese weiteren Vergleiche machen dann die restlichen Einträge in den Wertetupeln des Set von Wertetupeln aus. Gibt es dagegen keine Vergleiche außer den Gleichheitsvergleichen, so haben die Wertetupel im Set von Wertetupeln nur zwei Einträge. Beide Einträge wären dann Gleichheitsvergleiche.

In einer früheren Programmversion waren die Gleichheitsvergleiche jeweils extra pro Atomvariable die auf Gleichheit testet. Es ist aber effizienter diese in einer erstellten Werteliste `ImmutableList<Value<?>>` zu haben, da man so sofort (in $O(1)$) die Schnittmenge der Matches beider Atomvariablen erhält. So muss man nicht einzeln die Matches bilden (in $O(1)$) und von diesen dann die Schnittmenge (in $O(n)$, wobei n die größe des kleineren der beiden Ergebnis-Sets ist).

Test auf Ungleichheit

Der Test auf Ungleichheit wird durch die Variable `reteEntitiesLookupUnequal` durchgeführt:

```
final ImmutableMap<String, UnequalLookup>
   reteEntitiesLookupUnequal;
```

Wie an der Datenstruktur erkennbar ist, wird ein Nachschlagewerk pro Atomvariable gebildet. Das Nachschlagewerk selber ist `UnequalLookup`. Es funktioniert ähnlich wie der Gleichheitsvergleich mit einer Hashmap. Um den Ungleichheitsvergleich auszuführen muss zu beginn der Wert mit dem es nicht gleich sein darf maskiert werden. Dies wird durch `maskValue(Value<?> val)` erledigt. Wenn das matching dann abgeschlossen ist wird der `UnequalLookup` mit der Methode `repair()` wieder zurückgesetzt.

Für das Matching in `getMatches(...)` wird zusätzlich die Methode `getValuesCopy()` bereitgestellt. Diese wird benötigt, wenn aus dem `UnequalLookup` ein Set, aus allen enthaltenen Ergebnis-Wertetupeln, erstellt werden muss. Das ist nur der Fall wenn `UnequalLookup` die

kleinste Anzahl an Ergebnis-Wertetupeln aller Nachschläge liefert. Um die Anzahl von Ergebnis-Wertetupel zu verfolgen wird die Variable `size` definiert. Sie wird bei jedem `add(...)`, `remove(...)`, `maskValue(...)` und `repair()` aktualisiert. So kann man einen Aufruf auf `size()` in $O(1)$ durchführen, anstatt bei jedem Aufruf die größe jedes Sets in `lookupUnequal` neu addieren zu müssen. Das ist auch der Grund für die Subklasse `UnequalLookup`.

Ein Optimierung die nicht implementiert wurde ist, dass man das `UnequalLookup` pro Attribut zusammenführen kann. Also z.B. „`^value <> <atomVar1> ^value <> <atomVar2>`“, da beide das gleiche Attribut testen. Es würde in solchen Fällen den Speicherverbrauch verringern und die Matching-Geschwindigkeit erhöhen.

Test auf Typengleichheit

Der Test auf Typengleichheit wird durch die Variable `reteEntitiesLookupEnum` durchgeführt:

```
final ImmutableMap<String, EnumMap<ValueType,  HashSet<
   ImmutableList<Value<?>>>>> reteEntitiesLookupEnum;
```

Es enthält eine `EnumMap`, die Wertetypen auf alle Wertetupel mit jenem Typ abbildet. Alle Operationen `add(...)`, `remove(...)` und `getMatchesEnum(...)` sind dabei $O(1)$. Für verbesserte Effizienz könnten mehrere Typvergleiche wie beim Gleichheitsvergleich zusammengeführt werden. Das wurde aber nicht implementiert. Um es dann nochmal zu verbessern hätte man noch den vereinten Typvergleich mit dem vereinten Gleichheitsvergleich in gleicher Art zusammenführen können. Ein Typvergleich ist so gesehen nur eine besondere Art eines Gleichheitsvergleichs.

Größenvergleiche

Größenvergleiche werden durch die Variable `reteEntitiesLookupTree` durchgeführt:

```
final ImmutableMap<String, TreeLookup> reteEntitiesLookupTree;
```

Wie im Ungleichheitsvergleich, wird ein Nachschlagewerk pro Atomvariable gebildet. Das Nachschlagewerk selber ist in diesem Fall `TreeLookup`. Innerhalb von `TreeLookup` befindet sich der eigentliche `lookupTree`:

```
SetMultimap<Value<?>, ImmutableList<Value<?>>> lookupTree =
   MultimapBuilder.treeKeys().hashSetValues().build();
```

Es enthält ein `SetMultimap`, welches ähnlich dem Java `TreeSet` ist, nur dass für jeden Schlüssel mehrere Werte in einem `HashSet` gespeichert werden anstatt nur eines Werts. Es ist also ein balancierter Binärbaum mit `HashSet`’s als Blättern.

Auch ähnlich dem Ungleichheitsvergleich kann im `TreeLookup` ein Wert maskiert werden mit `maskValue(Value<?> val, boolean inclusive)` was nach dem Matching wieder mit `repair()` repariert wird. Wenn der Vergleich ein „=“ enthält, also bei „<=“ und „>=“, so ist `boolean inclusive` wahr. Wenn das „=“ fehlt, ist es falsch. Auf diese Art können alle verschiedenen Größenvergleiche durch maskieren oder nicht maskieren ausgeführt werden. Wenn maskiert wird, dann kostet das $O(log(treeSize))$ wobei `treeSize` die Anzahl der einzigartigen Werte der Atomvariable ist.

Ein Wertetupel mit `add(...)` hinzuzufügen oder mit `remove(...)` zu entfernen kostet $O(log(treeSize))$ Laufzeit. Um mit `TreeLookup` zu matchen wird entweder

```
tailSet(ImmutableList<Value<?>> values, boolean inclusive)}
```

für „>=“ und „>“ oder

```
headSet(ImmutableList<Value<?>> values, boolean inclusive)
```

für „<=“ und „<“ aufgerufen. Diese Aufrufe sind $O(1)$, ein späterer Aufruf auf `.size()` wird damit aber zu $O(log(treeSize))$.

Da nur numerische Werte verglichen werden dürfen, werden alle Matches mit einem nichtnumerischen Wert schon in `getMatches(...)` ausgefiltert. Das macht alle Größenvergleiche mit nichtnumerischen Werten $O(1)$.

Da in `getMatches(...)` die Teilmatches der einzelnen Nachschlagewerke nacheinander verglichen werden um die Schnittmenge zu bilden, wird auch die Ergebnis-Menge des `TreeLookup` verwendet. Dieses kann auf zwei Arten verwendet werden. Entweder es wird das Vorhandensein jedes Elements einzeln mit `boolean contains(ImmutableList<Value<?>> valueList)` getestet, was $O(log_2(treeSize))$ kostet oder es wird der ganze `TreeLookup` mit `getValuesCopy()` in sein ensprechendes Set umgewandelt, was $O(treeSize)$ Laufzeit kostet, danach aber bei jedem Nachschlag nur $O(1)$ ist.

Genauso wie im Code, seinen nun `res` die Elemente deren Vorhandensein und Nichtvorhandensein im `SetMultimap` getestet werden soll. `res.size()` ist entsprechend die Anzahl dieser Elemente. Das `SetMultimap` heißt im folgenden `set` und die größe dessen Baums (nicht die Totale Größe, da die Blatt-HashSet Nachschäge $O(1)$ sind) `set.uniqueKeysSize()`. Dann sind die Gleichungen für die Laufzeiten der zwei Fälle:

1. $O(res.size() * log_2(set.uniqueKeysSize()))$
   für das individuelle Nachschlagen mit `SortedSet.contains(...)`
2. $O(res.size() + set.uniqueKeysSize())$
   für das umwandeln des Baums in ein `HashSet` und dann Nachschlagen mit `HashSet.contains(...)`

Im Code wird immer die schnellere der zwei Vorgehensweisen gewählt. Das ist in [Fra24, Dateipfad: MemoReteOPS5/src/main/ops5/workingmemory/data/ValueListCompare.java getMatches(...)] im `switch (mapType)` Fall `TREE` einsehbar. Für die Berechnung des 2er Logarithmus wird ein schneller Binärcode verwendet: `public static int binlog(int bits)`.

Abbildung 5.5 zeigt die unterschiedlichen Laufzeiten der zwei Ansätze. Die Laufzeit ist an der vertikalen Achse. Die unteren Achsen sind die Größen der Mengen. Der erste Fall entspricht der blauen gekrümmten Fläche. Der zweite Fall entspricht der roten glatten Fläche. Die tatsächliche Laufzeit entspricht dann dem Minimum der zwei Flächen. Man sieht deutlich, dass sich meistens der zweite Fall mit der roten glatten Fläche lohnt. Nur wenn `res.size()` viel kleiner als `set.uniqueKeysSize()` oder wenn `set.uniqueKeysSize()` sehr klein ist, lohnt sich der erste Fall.

Zusammenführen mehrerer Größenvergleiche Wenn es mehrere Größenvergleiche gibt kann man diese zusammenlegen. Dies kann man z.B. mit einer Hilbertkurve [Wik09] oder mit der etwas einfacheren Z-Kurve [Wik17] bewerkstelligen. Dabei wird eine Dimension für jeden Größenvergleich angelegt. Beide Kurven erstellen aus Werten mehrerer Dimensionen (in MemoRete OPS5 wären dass die Atomvariablen die auf Größe vergleichen) einen Wert. Dieser neue Wert würde dann zum erstellen eines Binärbaums verwendet werden. Bei der Suche in dem Baum müsste aber dann ein passender Suchalgorithmus gefunden werden. Ein UB-Baum löst dieses Problem für Z-Kurven. Im Wikipedia Artikel dazu [Wik06b] steht:

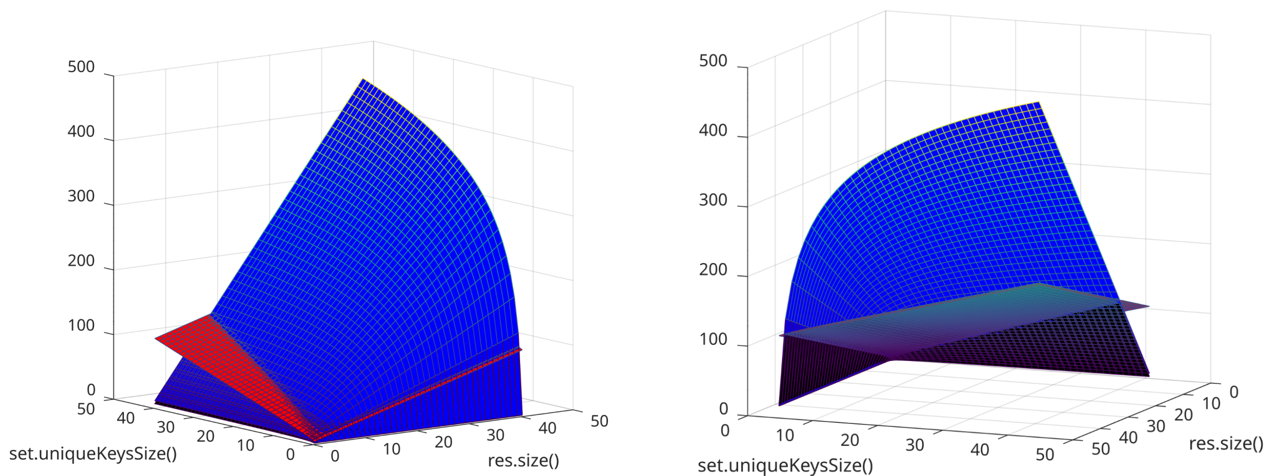

Abbildung 5.5: Laufzeit JOIN

> „Das hierfür ursprünglich von Rudolf Bayer angegebene Verfahren war im Aufwand exponentiell mit der Anzahl der Dimensionen und somit für mehr als 4 Dimensionen nicht praktisch verwendbar [Mar00]. Eine Lösung für das Problem ('crucial part of the UB-tree range query'), linear mit der Bitlänge der Z-Werte, wurde später beschrieben [Ram+00] ('GetNextZ-address'); diese Methode war bereits beschrieben worden in [TH81] ('BIGMIN'-Berechnung).“

In der „BIGMIN-Berechnung“ [TH81, S. 77] steht weiter, dass die Suche in $O(k(log(N)+F))$ Laufzeit stattfindet. Wobei k die Anzahl der Dimensionen, N die größe des Sets und F die Menge aller gefundenen Ergebnisse im Set ist [TH81, S. 77]. Im langsamsten Fall sind alle Elemente des Sets ein Ergebnis, also $F = N$. Damit gilt $O(k(log(N)+N))$. Vergleicht man das mit der in MemoRete OPS5 implementierten Suche, welche bei k Größenvergleichen $O(k * log(N))$ ist, so ist das fast gleich schnell. In MemoRete muss aus diesen Ergebnissen aber dann noch die Schnittmenge gebildet werden, was im langsamsten Fall $O(2*N)$ pro Schnittmenge ist plus noch ein $O(N)$ für die Umwandlung der kleinsten Ergebnismenge in ein HashSet. Damit dauert das Ergebnis zu bilden in MemoRete im langsamsten Fall $O(k*log(N)+2N(k-1)+N))$ was sich zu $O(k*log(N)+N(2k-1)))$ kürzt.

Die erwartete durchschnittliche Laufzeit in der „BIGMIN-Berechnung“ [TH81, S. 77] ist $O(k(log(N)+F))$. Weiter wird dort erwähnt, dass es sich um eine Annahme handelt. Es ist also unklar, ob sie wirklich gilt. In MemoRete OPS5 ist die Laufzeit im besten Fall $O(k(log(N))+k(N+F)+F)$ was wenn mans aufs wesentliche reduziert $O(k(N+F))$ ist. Diese Laufzeit tritt nur ein wenn eine Teilschnittmenge zufällig genau gleich der Ergebnismenge ist. Es würde sich also lohnen eine Z-Kurve in MemoRete OPS5 zu verwenden.

Weitere Arten von Größenvergleichen Man kann den Baumvergleich parallelisieren. Es gibt fertige Bibliotheken wie [Yih24], die das unterstützen und Operationen auf Bäumen maximal optimieren. In dem Wikipedia Artikel „Join-based tree algorithms“ steht [Wik18, übersetzt]:

> „Die Komplexität jeder Vereinigung, Schnittmenge und Differenz ist $O(m * log(n/m+1))$ für zwei balancierte Bäume mit den größen m und $n(>= m)$. Diese Komplexität ist im Hinblick auf die Anzahl der Vergleiche optimal. Noch wichtiger ist, dass die rekursiven Aufrufe von Vereinigung, Schnittmenge oder Differenz unabhängig voneinander ausgeführt werden können und daher parallel mit einer parallelen Tiefe $O(log(m) * log(n))$ ausgeführt werden können.“

Für jene Komplexität der Schnittmenge $O(m * log(n/m + 1))$ setzt man $m = n$, nimmt also gleiche Baumgröße an, so erhält man $O(n * log(2))$. Dagegen ist in MemoRete OPS5 eine Schnittmengenoperation zwischen zwei n großen Sets im langsamsten Fall $O(n * 3)$. Wenn man diese Art der Baum Schnittmengenbildung auch nach einem Aufruf ähnlich dem Java `headSet(...)` oder `tailSet(...)` mit gleicher Geschwindigkeit ausführen kann, dann wäre das eine Verbesserung für MemoRete.

5.6.9 Terminierungs-Knoten

Im Terminierungs-Knoten werden die Rete Einträge weitergeleitet an das Konfliktset. Das ist $O(1)$. Das Einsortieren ins Konfliktset ist allerdings $O(log_2(conflictSize))$, weil die Einträge in einem `TreeMultiset` gespeichert werden. Man könnte auch die Elemente im Konfliktset lose speichern und nur beim feuern alle Elemente nach dem zu feuernden durchsuchen. Das braucht $O(conflictSize)$ Laufzeit. Je nachdem wie viele Elemente in einem Recognize-Act-Zyklus zum Konfliktset hinzukommen ist eines von beiden schneller.

Die Gleichung $n * log_2(conflictSize) = conflictSize$ definiert das Equilibrium beider Ansätze, wobei n die Menge an neu dazugekommenen Elementen im Konfliktset pro Zyklus ist. Ist $n > conflictSize/log_2(conflictSize)$ so würde sich das einmalige Sortieren vor dem Feuern mehr lohnen. Ist $n < conflictSize/log_2(conflictSize)$ so lohnt sich der im Code verwendete Ansatz mit dem `TreeMultiset` mehr. Da Rete darauf baut, dass die Änderungen in der Faktenbasis pro Recognize-Act-Zyklus nur gering sind, sind vermutlich auch die Änderungen im Konfliktset kleiner als n. Ob das in der Praxis stimmt wurde allerdings nicht überprüft. Es kann jedenfalls von einem OPS5 Programm zum nächsten variieren.

Würde man als Regelfeuerstrategie eine Zufallsauswahl benutzen, so wäre das Einfügen ins Konfliktset in jedem Fall bei $O(1)$ Laufzeit. Zusammen mit einer „trägen“ Propagierung, ermöglicht durch die Zufallsauswahl, könnte man sehr schnelle Rete-Algorithmen implementieren.

5.7 Aktionsteil

5.7.1 Aktion Make, Modify, Remove

Die Aktionen Make, Modify und Remove verändern die Faktenbasis. In den Auswirkungen auf die Laufzeit sind sie eher gering verglichen mit der Matching-Phase. Atomvariablen und Elementvariablen müssen zwar, um deren Werte auszulesen, rekursiv den Baum nach oben gehen. Das ist aber bei normaler Regellänge nicht weit. Bei fünf Bedingungen im Bedingungsteil wäre die Tiefe höchstens 5 Schritte plus den Schritten der Teilbedingung in der obersten Bedingung, welche in der Regel auch im einstelligen Bereich liegen. Für die Laufzeit ist der Aufwand zum Feuern der Aktionen Make, Modify und Remove gering.

5.7.2 Aktion Call

Anders sieht es dagegen bei Aktion Call aus, denn diese blockiert so lange der Code, der „gecallt“ wird, dauert. Des Weiteren muss bei Aktion Call die richtige überladene Methode oder Konstruktor gefunden werden. Zur instantiierung von `ActionCall.java` werden schon alle Methoden und Konstruktoren, die aufgrund des Aufrufschemas nicht in frage kommen, ausgefiltert in

```
ArrayList<Method> prefilterMethods(Method[] methods, String[]
   parameters, String methodName) ...
```

und

```
ArrayList<Constructor<?>> prefilterConstructors(Constructor<?>[]
    constructors, String[] parameters) ...
```

Unter den übrig gebliebenen wird dann dynamisch zu Laufzeit mit entweder `getMatchingMethod(...)` oder `getMatchingConstructor(...)` die passende Methode oder der passende Konstruktor gewählt.

Anzumerken ist, dass gerade dieses blockierende Verhalten bei den Call Aufrufen es schwierig machen kann einen Roboter interaktiv zu steuern. Man müsste entweder sicherstellen, dass alle Aufrufe schnell abgeschlossen sind oder in MemoRete das unabhängige gleichzeitige Ausführen mehrerer OPS5 Programme ermöglichen.

Die Subaktionen Make, Modify und Remove sind ähnlich den Aktionen oben beschrieben.

5.7.3 Funktion Compute

Die Funktion Compute erstellt zu seiner Funktion einen rekursiven Baum aus `ComputeEntryType`. Bei der Ausführung der Funktion Compute wird dieser Baum auch rekursiv mit einem Stapelspeicher `Stack<CalculatorTriplet> stackCalculators` abgegangen und mitfilfe des jeweils aktuellen `CalculatorTriplet` die Werte berechnet.

Für die Laufzeit ist das nicht sehr relevant, da die Rekursion nur so tief ist, wie der Benutzer Klammern im OPS5 Programm setzt. Man könnte aber zum Beispiel Eingaben wie `(Compute 1+1+<atomX>)` auf `(Compute 2+<atomX>)` kürzen. Für die Laufzeit ist das aber nicht relevant und wurde so nicht implementiert.

In Compute werden auch Atomvariablen für die Berechnung benutzt. Das schränkt den möglichen Typ der Atomvariable auf `Double` und `Integer` ein. Ist kein numerischer Typ in der Atomvariable gespeichert, so gibt Compute `NIL` als Wert zurück. Des Weiteren ist ein Teilen durch 0 undefiniert.

Kapitel 6

Schluss

In dieser Arbeit wurden viele verschiedene Möglichkeiten des Rete Algorithmus abgewägt anhand einer neuen Implementierung namens MemoRete OPS5. Das Prinzip der Memoisation, die Indizierung für Variablenvergleiche, das Benutzen von Wertetupeln und die Verwendung von Multisets haben sich als geeignete Mittel zum schnellen und platzeffizienten Matching herausgestellt. Es wurde festgestellt, dass das Matching große Ähnlichkeit mit der JOIN Operation in Datenbanken hat und man so einige Verbesserungen und Forschungsergebnisse aus den Datenbanken wiederverwenden kann.

Die Untersuchung war aber nicht erschöpfend. Und so ist der MemoRete Algorithmus auch noch nicht optimal. Es wäre interessant wie sich MemoRete im Vergleich zu den neueren Rete Varianten, Rete II oder Rete-NT verhält. Sicher ist aber, dass mit MemoRete ein Schritt in die richtige Richtung getan ist. Das Regelmatching birgt auch noch 45 Jahre nach der Veröffentlichung des originalen Rete-Algorithmus Chancen auf Verbesserung. Man darf gespannt sein was sich ambitionierte Informatiker und Forscher noch einfallen lassen.

Literatur

[CF00] Albert Mo Kim Cheng und Seiya Fujii. “Bounded-response-time self-stabilizing OPS5 production systems”. In: *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*. IEEE. 2000, S. 399–404.

[Fra24] Lukas Frank. *MemoRete OPS5 Quellcode*. 2024. URL: `https://github.com/LukasF1337/MemoReteOPS5`.

[GR98] Joseph Giarratano und Gary Riley. *Expert systems, Principles and Programming*. 1998. ISBN: 7-111-10844-2/TP.2586.

[Höf+09] Otfried Höffe u. a. *Aristoteles: die Hauptwerke: ein Lesebuch*. Gunter Narr Verlag, 2009.

[Kri87] Reinhard Krickhahn. *Die Wissensrepräsentationssprache OPS5: Sprachbeschreibung und Einführung in die regelorientierte Programmierung*. Springer-Verlag, 1987. ISBN: 978-3-528-04498-5.

[Mar00] Volker Markl. *Mistral: Processing relational queries using a multidimensional access technique*. Springer, 2000.

[Owe10] James Owen. *World’s fastest rules engine*. 2010. URL: `https://www.infoworld.com/article/2078197/world-s-fastest-rules-engine.html`.

[Ram+00] Frank Ramsak u. a. “Integrating the UB-tree into a database system kernel.” In: *VLDB*. Bd. 2000. 2000, S. 263–272.

[TH81] Herbert Tropf und Helmut Herzog. “Multidimensional Range Search in Dynamically Balanced Trees.” In: *ANGEWANDTE INFO.* 2 (1981), S. 71–77.

[VVP02] Richard G Vedder, Thomas P Van Dyke und Victor R Prybutok. “Death of an expert system: A case study of success and failure”. In: *Journal of International Information Management* 11.1 (2002).

[Wik04] Wikipedia. *Scholastik*. [Online; Zugriff am 11.01.2024]. 2004. URL: `https://de.wikipedia.org/wiki/Scholastik#Deduktives_Prinzip`.

[Wik06a] Wikipedia. *Materialized view*. [Online; Zugriff am 08.01.2024]. 2006. URL: `https://en.m.wikipedia.org/wiki/Materialized_view`.

[Wik06b] Wikipedia. *UB-Baum*. [Online; Zugriff am 10.01.2024]. 2006. URL: `https://de.wikipedia.org/wiki/UB-Baum`.

[Wik09] Wikipedia. *Hilbert-Kurve*. [Online; Zugriff am 09.01.2024]. 2009. URL: `https://de.wikipedia.org/wiki/Hilbert-Kurve`.

[Wik17] Wikipedia. *Z-Kurve*. [Online; Zugriff am 07.01.2024]. 2017. URL: `https://de.wikipedia.org/wiki/Z-Kurve`.

[Wik18] Wikipedia. *Join-based tree algorithms*. [Online; Zugriff am 09.01.2024]. 2018. URL: `https://en.m.wikipedia.org/wiki/Join-based_tree_algorithms`.

[Yih24] Daniel Ferizovic Yihan Sun Guy E. Blelloch. *PAM (Parallel Augmented Maps)*. [Online; Zugriff am 06.01.2024]. 2024. URL: https://cmuparlay.github.io/PAMWeb/.

Erklärung

Ich erkläre hiermit, dass ich die Arbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benützt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Schrobenhausen, 12. Januar 2024

(Unterschrift)
Vorname, Name

Anhang A

Anlage digitale Medien

Mit zur Bachelorarbeit wurden folgende Dateien abgegeben:

- Ordner `MemoReteOPS5/` enthält das MemoRete OPS5 Java Programm. Es ist auch Online zum Download erhältlich auf der Seite:
  `https://github.com/LukasF1337/MemoReteOPS5`
- Datei `tree_intersect_graph.m` ist ein GNU Octave Programm zur Erstellung der Abbildung 5.5.