UNI KASSEL VERSITÄT | ELEKTROTECHNIK INFORMATIK

Eine Erweiterung des DiMo-Tools um andere Datentypen – Am Beispiel von Wörtern

Bachelorarbeit
am Fachgebiet Theoretische Informatik / Formale Methoden
der Universität Kassel

Autor: Nathanael Schmidt

Erstprüfer: Prof. Dr. Martin Lange
Zweitprüferin: Prof. Dr. Claudia Fohry
Betreuer: Dr. Norbert Hundeshagen / M.Sc. Maurice Herwig

Eingereicht am: 10. Januar 2025

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst habe und alle verwendeten Quellen und Hilfsmittel angegeben habe.

Nathanael Schmidt

Kassel, 10. Januar 2025

Inhaltsverzeichnis

1 Einleitung **4**
1.1 Anforderungen 4

2 Grundlagen **5**
2.1 Mathematische Grundlagen 5
2.2 Grundlagen von DiMo 9

3 Erweiterung der DiMo-Sprache **11**
3.1 Syntaktische und semantische Erweiterungen 12
3.2 Implementierung der Erweiterungen 21

4 Anwendungsfälle **29**
4.1 Teilwort-Problem 29
4.2 Teilfolgen-Problem 31
4.3 Präfix-Problem 32
4.4 Levenshtein-Distanz-Problem 33

5 Fazit **37**
5.1 Offene Probleme 37
5.2 Mögliche Erweiterungen 38

Literatur **39**

Verzeichnis der Listings

1 Header im DiMo-Code 9
2 Formelbereich im DiMo-Code 10
3 Ausgabe der Ausführung von DiMo auf dem Beispiel-Code 11
4 Header eines DiMo-Programms mit Wörtern 13
5 Formelbereich eines DiMo-Programms mit Wörtern 13
6 Ausschnitt aus den Ableitungen des Nichtterminals `domain` 23
7 Der Typ `intTerm` vorher 26
8 Der Typ `term` jetzt 26
9 Der Typ `symbSet` vorher 26
10 Der Typ `symbSet` jetzt 26
11 Die Funktion `evalTerm` jetzt 27
12 Hier wird das implizite/explizite Alphabet festgelegt 29
13 Ausgabe der Ausführung von DiMo auf dem Teilwort-Problem 30
14 Ausgabe der Ausführung von DiMo auf dem Teilfolgen-Problem 31
15 Ausgabe der Ausführung von DiMo auf dem Präfix-Problem 32
16 Ausgabe der Ausführung von DiMo auf dem Levenshtein-Distanz-Problem 36

Abbildungsverzeichnis

1 Benutzeroberfläche von DiMo Tool Web 12

Algorithmenverzeichnis

1 Naive Aufzählung von Sprachen regulärer Ausdrücke 14

1 Einleitung

Aussagenlogik ist ein nützliches Werkzeug, womit sich viele Probleme aus der Mathematik und Informatik beschreiben lassen. Z. B. lassen sich der Aufbau boolescher Schaltkreise oder künstlicher Intelligenzen aussagenlogisch modellieren [1]. Wurde ein Problem in Aussagenlogik modelliert, kann mit einer speziellen Software, einem SAT-Solver, überprüft werden, ob die Formel erfüllbar (bzw. allgemeingültig) ist. Ist dies der Fall, bedeutet das, dass das Problem auch lösbar ist. Dann enthält die Ausgabe des SAT-Solvers oft auch die Information, *wie* das Problem gelöst werden kann. Da viele Probleme und somit deren zugehörige Formeln von mehreren Parametern abhängig sind, bietet es sich an, sogenannte parametrisierte Formeln anzugeben. Aus diesen können dann konkrete Formeln erstellt werden, wenn den Parametern Werte zugewiesen worden sind. Hierfür wurde das Tool DiMo [2] entwickelt.
Mit diesem Tool ist es möglich, parametrisierte aussagenlogische Formeln zu beschreiben und diese anschließend für verschiedene Parameter auf Erfüllbarkeit, Allgemeingültigkeit oder semantische Äquivalenz zu überprüfen oder alle Modelle der Formel auszugeben. Allerdings konnten die Formelparameter in DiMo bisher nur natürliche Zahlen sein. Da es aus Sicht der zu modellierenden Probleme sinnvoll ist, andere Typen zu haben (z. B. Wörter oder Graphen), sollte in dieser Arbeit nun DiMo um Wörter als Datentyp erweitert werden. Aufbauend darauf soll es auch möglich sein, weitere Datentypen zu implementieren.

1.1 Anforderungen

Wenn ein neuer Datentyp hinzugefügt werden soll, muss definiert werden, wie dieser verwendet werden kann. Hierfür muss eine Menge M festgelegt werden, deren Elemente durch Instanzen dieses Datentyps repräsentiert werden können. Im Fall von Wörtern soll dies die Menge aller Wörter über einem Alphabet sein. Definitionsbereiche von Parametern und Iterationsmengen bei verallgemeinerten Junktoren sind dann immer Teilmengen dieser Menge. Hierbei muss man sich überlegen, welche Teilmengen wie beschrieben werden sollen. Für Wörter sollen das zum einen endliche Mengen sein, bei denen alle Elemente explizit aufgezählt werden, und zum anderen reguläre Sprachen, die durch reguläre Ausdrücke angegeben werden. Dabei ist zu beachten, dass solche Teilmengen aufzählbar sein müssen, da DiMo für die Instanziierung der Formel die Menge aufzählt. Zu den Anforderungen gehört also auch ein Algorithmus, um die Teilmengen aufzuzählen, was insbesondere für unendliche Mengen nicht trivial ist. Außerdem muss noch eine Menge von Operationen definiert werden, die auf diesem Datentyp ausgeführt werden können. Folgende Operationen soll es für Wörter geben:

- Bildung der Länge des Wortes, welche vom Typ eine Ganzzahl ist
- Zugriff auf Zeichen eines Wortes
- Bildung von Teilwörtern
- Konkatenation von Wörtern
- Mehrfache Konkatenation eines Wortes, geschrieben als Potenz dieses Wortes zu einer Ganzzahl

2 Grundlagen

In den folgenden Absätzen werden die für diese Arbeit nötigen Grundlagen erläutert. Dies umfasst mathematische Notationen sowie den Aufbau von DiMo.

2.1 Mathematische Grundlagen

Ein Großteil der mathematischen Definitionen ist (teilweise etwas abgeändert) aus [3] entnommen.
Da DiMo mit aussagenlogischen Formeln arbeitet, muss zuerst definiert werden, wie solche Formeln aufgebaut sind.

Definition 1. Sei Var eine abzählbare Menge von Variablen. Dann ist die Menge $\mathrm{AL}(Var)$ aussagenlogischer Formeln über Var die kleinste Menge, für die gilt:

(1) $\mathtt{True}, \mathtt{False} \in \mathrm{AL}(Var)$

(2) Für alle $A \in Var$ gilt: $A \in \mathrm{AL}(Var)$

(3) Für aussagenlogische Formeln $\varphi, \psi \in \mathrm{AL}(Var)$ gilt:

- $\neg\varphi \in \mathrm{AL}(Var)$
- $(\varphi \wedge \psi) \in \mathrm{AL}(Var)$
- $(\varphi \vee \psi) \in \mathrm{AL}(Var)$
- $(\varphi \rightarrow \psi) \in \mathrm{AL}(Var)$
- $(\varphi \leftrightarrow \psi) \in \mathrm{AL}(Var)$

Beispiel 1. Eine Variablenmenge kann z. B. so aussehen:

$$Var = \{A, B, C_1, C_2, D_1, D_2, \dots, D_n, \dots\}$$

Eine mögliche Formel über Var ist dann:

$$\varphi = (((A \wedge B) \vee C_1) \leftrightarrow (\mathtt{True} \wedge D_2))$$

Definition 2. Für eine endliche Menge $\Phi = \{\varphi_1, \dots, \varphi_n\} \subseteq \mathrm{AL}(Var)$ seien die **verallgemeinerten Junktoren** definiert als:

$$\bigwedge \Phi = \bigwedge_{i=1}^{n} \varphi_i = \varphi_1 \wedge \dots \wedge \varphi_n$$

$$\bigvee \Phi = \bigvee_{i=1}^{n} \varphi_i = \varphi_1 \vee \dots \vee \varphi_n$$

Die Menge Φ wird im Folgenden auch als **Iterationsmenge** des verallgemeinerten Junktors bezeichnet.

Bisher wurde nur die Syntax von Formeln betrachtet. Nun muss noch die Semantik von Formeln festgelegt werden:

Definition 3. Sei $\mathcal{I}\colon Var \rightarrow \{0, 1\}$ eine Abbildung für eine Variablenmenge Var. Diese wird induktiv für $\varphi, \psi \in \mathrm{AL}(Var)$ folgendermaßen definiert:

- $\mathcal{I}(\texttt{True}) = 1$
- $\mathcal{I}(\texttt{False}) = 0$
- $\mathcal{I}(\neg\varphi) = \begin{cases} 1, & \text{falls } \mathcal{I}(\varphi) = 0 \\ 0, & \text{sonst} \end{cases}$
- $\mathcal{I}(\varphi \wedge \psi) = \begin{cases} 1, & \text{falls } \mathcal{I}(\varphi) = 1 \text{ und } \mathcal{I}(\psi) = 1 \\ 0, & \text{sonst} \end{cases}$
- $\mathcal{I}(\varphi \vee \psi) = \begin{cases} 1, & \text{falls } \mathcal{I}(\varphi) = 1 \text{ oder } \mathcal{I}(\psi) = 1 \\ 0, & \text{sonst} \end{cases}$
- $\mathcal{I}(\varphi \rightarrow \psi) = \begin{cases} 1, & \text{falls } \mathcal{I}(\varphi) = 0 \text{ oder } \mathcal{I}(\psi) = 1 \\ 0, & \text{sonst} \end{cases}$
- $\mathcal{I}(\varphi \leftrightarrow \psi) = \begin{cases} 1, & \text{falls } \mathcal{I}(\varphi) = \mathcal{I}(\psi) \\ 0, & \text{sonst} \end{cases}$

Definition 4. Eine Interpretation $\mathcal{I}$ ist **Modell** einer Formel $\varphi \in \mathrm{AL}(\mathit{Var})$, genau dann wenn $\mathcal{I}(\varphi) = 1$ gilt.
Man schreibt dann auch: $\mathcal{I} \models \varphi$

Definition 5. Eine Formel $\varphi \in \mathrm{AL}(\mathit{Var})$ bezeichnet man als…

- …**erfüllbar**, falls es eine Interpretation gibt, die Modell von φ ist.
- …**allgemeingültig**, falls jede Interpretation Modell von φ ist.
- …**unerfüllbar**, falls keine Interpretation Modell von φ ist.

Definition 6. Zwei Formeln $\varphi, \psi \in \mathrm{AL}(\mathit{Var})$ sind **semantisch äquivalent**, wenn für alle Interpretationen $\mathcal{I}$ gilt: $\mathcal{I}(\varphi) = \mathcal{I}(\psi)$

Nachdem die Aussagenlogik formal definiert wurde, müssen nun als Nächstes noch Wörter und deren Operationen definiert werden, da diese zu DiMo hinzugefügt werden sollen.

Definition 7. Ein **Wort** ist eine Abbildung $w\colon \{1, \ldots, n\} \rightarrow \Sigma$, wobei Σ eine endliche, nicht-leere Menge von Zeichen ist. $w(i)$ wird als Zeichen an der Stelle i von w bezeichnet. $n \in \mathbb{N}$ wird als **Länge** von w bezeichnet, man schreibt auch $|w| = n$. Ist $n = 0$, enthält w keine Zeichen, w ist das **leere Wort**. Das leere Wort wird auch mit ε bezeichnet.

Man kann ein Wort w auch als eine Folge von Zeichen verstehen, man schreibt dann $w = w_1 w_2 \ldots w_n$, wobei $w_i = w(i)$.

Definition 8. Sei $w = w_1 w_2 \ldots w_n \in \Sigma^*$ ein Wort über dem Alphabet Σ. Dann sei für $i, j \in \mathbb{N}$ mit $1 \leq i, j \leq |w|$ das **Teilwort** $w(i, j)$ definiert als:

$$w(i, j) = \begin{cases} w_i \ldots w_j, & \text{falls } i \leq j \\ \varepsilon, & \text{sonst} \end{cases}$$

Definition 9. Seien $u, v \in \Sigma^*$ Wörter über einem Alphabet Σ. Dann ist die **Konkatenation** $u \cdot v \colon \{1, \dots, |u| + |v|\} \to \Sigma$ definiert über:

$$(u \cdot v)(i) = \begin{cases} u(i), & \text{falls } i \leq |u| \\ v(i - |u|), & \text{sonst} \end{cases}$$

Betrachtet man Wörter als Folgen von Zeichen, ist ihre Konkatenation also die Folge, die entsteht, wenn man diese Wörter hintereinander schreibt. Zum Beispiel ist für die Wörter $u = ab$ und $v = bc$ die Konkatenation $u \cdot v = abbc$. Mithilfe der Konkatenation lässt sich nun auch eine Potenz von Wörtern definieren.

Definition 10. Für ein Wort $w \in \Sigma$ und eine natürliche Zahl $n \in \mathbb{N}$ ist die n-te **Potenz** von w ein Wort, welches folgendermaßen induktiv definiert ist:

(1) $w^0 = \varepsilon$

(2) $w^n = w \cdot w^{n-1}$, für $n > 0$

Die n-te Potenz eines Wortes w lässt sich also auch so verstehen, dass w n-mal hintereinander geschrieben wird. Für $w = abbc$ gilt zum Beispiel $w^3 = (abbc)^3 = abbcabbcabbc$.

Da in DiMo auch Mengen von Werten angegeben werden müssen, muss nun noch definiert werden, wie Mengen von Wörtern beschrieben werden können und welche Operationen auf ihnen möglich sind.

Definition 11. Eine **Sprache** L über einem Alphabet Σ ist eine Menge von Wörtern über Σ.

Definition 12. Die **Konkatenation** $L_1 \cdot L_2$ zweier Sprachen $L_1, L_2 \subseteq \Sigma^*$ ist definiert als:

$$L_1 \cdot L_2 = \{u \cdot v \mid u \in L_1, v \in L_2\}$$

Definition 13. Die **kleenesche Hülle** L^* einer Sprache $L \subseteq \Sigma^*$ ist definiert als:

$$L^* = \{w_1 \cdot \ldots \cdot w_n \mid w_i \in L \text{ für } i \in \mathbb{N}\}$$

Die kleenesche Hülle einer Sprache L enthält also jede mögliche Folge von n Wörtern aus L. Im Fall $n = 0$ ist die Wortfolge leer, sodass das leere Wort herauskommt. Da dies unabhängig von L ist, enthält die kleenesche Hülle jeder Sprache das leere Wort. Außerdem lässt sich jede nicht-leere Wortfolge der Länge n als Konkatenation von einem Wort aus L und einer Wortfolge von $n - 1$ Wörtern aus L betrachten. Dies führt zu folgender äquivalenter Formulierung der kleeneschen Hülle:

Satz 14. *Für jede Sprache $L \subseteq \Sigma^*$ gilt:*

$$L^* = \{\varepsilon\} \cup \bigcup_{w \in L} w \cdot L^*$$

Dieser Satz lässt sich auch aus Ardens Lemma [4, S. 141-142] herleiten.

Definition 15. Die **positive Hülle** L^+ einer Sprache $L \subseteq \Sigma^*$ ist definiert als:

$$L^+ = L^* \setminus \{\varepsilon\}$$

Nachdem jetzt Sprachen und Operationen auf ihnen definiert sind, wird nun mit regulären Ausdrücken eine Möglichkeit gezeigt, wie einige Sprachen beschrieben werden können.

Definition 16. Die Menge der **regulären Ausdrücke** RegEx über einem Alphabet Σ ist die kleinste Menge für die gilt:

(1) $\emptyset \in \text{RegEx}$

(2) $\varepsilon \in \text{RegEx}$

(3) $\forall a \in \Sigma \colon a \in \text{RegEx}$

Für $\alpha, \beta \in \text{RegEx}$ gilt außerdem induktiv:

(4) $\alpha + \beta \in \text{RegEx}$

(5) $\alpha \cdot \beta \in \text{RegEx}$

(6) $\alpha^* \in \text{RegEx}$

Nachdem in Definition 16 die Syntax für reguläre Ausdrücke festgelegt wurde, wird nun ihre Semantik definiert:

Definition 17. Die Funktion $L \colon \text{RegEx} \to \mathcal{P}(\Sigma^*)$, die jedem regulären Ausdruck eine Sprache zuordnet, ist folgendermaßen definiert:

(1) $L(\emptyset) = \emptyset$

(2) $L(\varepsilon) = \{\varepsilon\}$

(3) $L(a) = \{a\}$

Sowie für $\alpha, \beta \in \text{RegEx}$:

(4) $L(\alpha + \beta) = L(\alpha) \cup L(\beta)$

(5) $L(\alpha \cdot \beta) = L(\alpha) \cdot L(\beta)$

(6) $L(\alpha^*) = L(\alpha)^*$

Bei DiMo wird für jeden Parameter der Formel eine Menge als Definitionsbereich angegeben, in der die Werte des Parameters liegen. Für solche Mengen sind in DiMo Aufzählungen implementiert. Wird ein neuer Datentyp hinzugefügt, müssen neue Definitionsbereiche mit ihren Aufzählungen implementiert werden. Hierfür folgt nun eine Definition von Aufzählungen:

Definition 18. Eine **Aufzählung** einer Menge M ist eine totale und berechenbare Funktion $f \colon \mathbb{N} \to M \cup \{\texttt{None}\}$ mit $f(\mathbb{N}) \supseteq M$, sodass für alle $n \in \mathbb{N}$ gilt:

$$f(n) = \texttt{None} \Rightarrow f(n+1) = \texttt{None}$$

O. B. d. A. sei `None` $\notin M$ angenommen.

Ist eine Funktion f eine Aufzählung von M, gibt es also für alle $m \in M$ ein $n \in \mathbb{N}$ mit $f(n) = m$. Man beachte, dass die hier verwendete Definition von Aufzählungen von der sonst üblichen abweicht, indem die Zielmenge noch ein Element `None` enthält, welches nicht in M enthalten ist. Daher ist f entgegen der üblichen Definition auch nicht zwingend surjektiv, da es für unendliche M kein n geben kann, für das $f(n) = \texttt{None}$ gilt. Ziel dieser Änderung ist, die Definition an die Implementierung in DiMo anzupassen, wo $f(n) = \texttt{None}$ bedeutet, dass alle Werte aus M bereits ausgegeben wurden und die Aufzählung von M abgeschlossen ist. Somit ist es möglich, dass f auch für endliche M insofern injektiv ist, dass es für alle $m \in M$ höchstens ein $n \in \mathbb{N}$ gibt, sodass $f(n) = m$ gilt. Im Folgenden wird f dann als injektiv auf M bezeichnet. Es gibt jedoch immer noch unendlich viele $n \in \mathbb{N}$ (in dem Fall alle $n \geq |M|$) mit $f(n) = \texttt{None}$, sodass f nicht wirklich injektiv ist.

2.2 Grundlagen von DiMo

Das DiMo-Tool ist ursprünglich ein Kommandozeilenprogramm, für das es mittlerweile mit der Webanwendung *DiMo Tool Web* auch eine grafische Benutzeroberfläche gibt [5], die im Hintergrund die Konsolenanwendung ausführt. In dieser Arbeit wurde die Erweiterung nur am Kommandozeilenprogramm vorgenommen, die Anpassung der Webanwendung hätte den Umfang der Arbeit überschritten. Zur Veranschaulichung der Funktionsweise von DiMo wird trotzdem in Abbildung 1 ein Screenshot von DiMo Tool Web gezeigt. Die Sprache von DiMo, die verwendet wird, um aussagenlogische Formeln auszudrücken, wird dabei an einem Beispiel verdeutlicht, welches im Quellcode von DiMo enthalten ist [6]. Die vollständige Grammatik ist in der Dokumentation von DiMo [2] zu finden.

```
1 SATISFIABLE Code(n)
2 PROPOSITIONS B
3 PARAMETERS n: {10}
```

Listing 1: Header im DiMo-Code

In Abbildung 1 ist links der DiMo-Code zu sehen, der in drei Bereiche eingeteilt ist. Die Aufteilung ist im Screenshot daran zu erkennen, dass am Anfang eines neuen Bereichs die Zeilennummerierung wieder bei 1 beginnt. Der erste Bereich wird im Folgenden als *Header* bezeichnet, der zweite als *Formelbereich* und der dritte als *Ausgabebereich*. Der Code im Header, der nochmal separat in Listing 1 zu sehen ist, beginnt mit `SATISFIABLE Code(n)`. Hiermit wird ausgedrückt, dass die nachfolgende Formel `Code` (mit Parameter `n`) auf Erfüllbarkeit getestet werden soll. Soll die Formel stattdessen auf Allgemeingültigkeit getestet werden, wird statt `SATISFIABLE` das Schlüsselwort `VALID` verwendet. Um nach allen Modellen für die Formel zu suchen, wird das Schlüsselwort `MODELS` verwendet. Mit `PROPOSITIONS B` wird ausgedrückt, dass `B` als Variablensymbol genutzt werden soll. Es ist auch möglich, mehrere Variablensymbole kommagetrennt nach dem Schlüsselwort `PROPOSITIONS` anzugeben. Mit dem Schlüsselwort `PARAMETERS` werden die Parameter der zu überprüfenden Formel festgelegt. Hierbei steht vor dem Doppelpunkt der Name des Parameters (in diesem Fall `n`) und hinter dem Doppelpunkt eine Menge, womit ausgedrückt wird, dass der Parameter immer ein Element dieser Menge ist. Diese wird im Folgenden als *Definitionsbereich* des Parameters `n` bezeichnet. In diesem Beispiel hat die Formel einen Parameter `n`, dessen Definitionsbereich nur den Wert 10 enthält. Definitionsbereiche können auch als Ganzzahlintervall angegeben werden (z. B. `{1,3,..,7}` für alle ungeraden Zahlen von 1 bis 7) und können sogar unendlich viele Werte enthalten (z. `42,44,..` für alle geraden Zahlen größer als 42).

```
1 FORMULAS
2
3 Code(n) =
4     FORALL i:{0,..,LOG (MAX {n,1})}.
5         (FORALL j:{1,.., (n / (2^i)) MOD 2}. B(i))
6        & FORALL j:{(n / (2^i)) MOD 2,..,0}. -B(i)
```

Listing 2: Formelbereich im DiMo-Code

Der Formelbereich des Codes ist in Listing 2 zu sehen. Dieser beginnt immer mit dem Schlüsselwort `FORMULAS`. In diesem Beispiel wird dort die Formel `Code` als parametrisierte Formel mit Parameter `n` definiert. Hier sieht man, dass DiMo einige mathematische Operationen unterstützt, wie die Grundrechenarten (inkl. Division mit Rest), Potenzen, Minimum/Maximum und Logarithmus (zur Basis 2). Die logischen Operatoren $\neg$, $\wedge$, $\vee$, $\rightarrow$ und $\leftrightarrow$ werden in DiMo mit den Symbolen `-`, `&`, `|`, `->` und `<->` dargestellt. Mit den Schlüsselwörtern `FORALL` und `FORSOME` werden die verallgemeinerten Junktoren $\bigwedge$ bzw. $\bigvee$ (s. Definition 2) dargestellt. Hinter diesem Schlüsselwort folgt der Name des Parameters und ein Doppelpunkt, auf den die Iterationsmenge des Junktors folgt. Die Schreibweise der Iterationsmengen ist sehr ähnlich zu der der Definitionsbereiche, mit dem Unterschied, dass hier keine unendlichen Mengen angegeben werden dürfen. Dafür müssen die Elemente aber keine Konstanten sein, sondern können mit den mathematischen Operationen aus Parametern berechnet werden. Zum Beispiel ist das letzte Element der Iterationsmenge des Junktors mit Parameter `i` abhängig vom Parameter `n`. Die Mengen können auch Vereinigungen, Schnitte oder Differenzen von mehreren Mengen sein. Außerdem ist hier zu sehen, dass Variablensymbole wie in Beispiel 1 auch parametrisiert sein können. Z. B. steht `B(i)` für die Variable B_i, wobei $i \in \mathbb{N}$ ein Parameter, des Variablensymbols B ist. Der DiMo-Code definiert also folgende Formel mit Parameter `n`:

$$\varphi_{\texttt{Code}}(n) = \bigwedge_{i=0}^{\lceil \log(\max\{n,1\}) \rceil} \left(\bigwedge_{j=1}^{\left\lfloor \frac{n}{2^i} \right\rfloor \bmod 2} B_i \right) \wedge \bigwedge_{j=\left\lfloor \frac{n}{2^i} \right\rfloor \bmod 2}^{0} \neg B_i$$

Im Folgenden werden Berechnungsausdrücke in DiMo, wie sie zur Mengenkonstruktion verwendet werden oder als Parameter an Unterformeln übergeben werden, als *Terme* bezeichnet. Zu beachten ist, dass Mengen oder logische Ausdrücke in Formeln *keine* Terme sind.

Beispiel 2. Folgende Ausdrücke aus Listing 2 sind Beispiele für Terme:

- `LOG (MAX n,1)`
- `n / (2^i)`
- `(n / (2^i)) MOD 2`

Folgendes sind keine Terme:

- `{0,..,LOG (MAX n,1)}` ist eine Menge
- `-B(i)` ist ein Formelschema

In Listing 3 ist die Konsolenausgabe von DiMo bei direkter Ausführung der Konsolenanwendung mit dem Flag `--debug` auf diesem Beispiel zu sehen. Bei der Ausführung wird

der Definitionsbereich $D_{\mathtt{n}}$ von `n` aufgezählt. Genauer gesagt wird für eine Aufzählung $f_{D_{\mathtt{n}}}$ für $D_{\mathtt{n}}$ ein Zähler $i \in \mathbb{N}$ hochgezählt und $f_{D_{\mathtt{n}}}(i)$ berechnet. Im Fall $f_{D_{\mathtt{n}}}(i) \neq \mathtt{None}$ gilt $f_{D_{\mathtt{n}}}(i) \in D_{\mathtt{n}}$ und die Formel $\varphi_{\mathtt{Code}}(n)$ kann mit $n = f_{D_{\mathtt{n}}}(i)$ instanziiert werden. Für jeden aufgezählten Wert gibt es in der Ausgabe einen Abschnitt, der mit der Ausgabe des Zählers i (hier mit `Running iteration 0` in Zeile 7) und des aufgezählten Wertes (Zeile 8) beginnt. Wegen $D_{\mathtt{n}} = \{10\}$ und $i = 0$ ergibt sich in diesem Fall $n = f_{D_{\mathtt{n}}}(i) = 10$. Nun wird $\varphi_{\mathtt{Code}}(n)$ mit $n = 10$ instanziiert. Diese Formel wurde in Zeile 9 ausgegeben, wobei in der Ausgabe die Wahrheitskonstanten `True` und `False` mit `tt` bzw. `ff` dargestellt werden. Nach der Instanziierung wird die Formel vereinfacht, indem u. a. überschüssige Wahrheitskonstanten aus der Formel entfernt werden (s. Zeile 10). Daraufhin muss die Formel noch in konjunktive Normalform (kurz *CNF*) [7, Abschnitt 3.6] gebracht werden, da der verwendete SAT-Solver Formeln in dieser Form erwartet. Die CNF dieser Formel ist in Zeile 12 zu sehen. Nun wird die Formel mit dem SAT-Solver auf Erfüllbarkeit getestet und das Ergebnis (Zeile 15) sowie das gefundene Modell (Zeile 18) ausgegeben. Der HTML-Code in Zeile 19 ist der einzige Teil der Ausgabe, der auch ohne Debug-Flag ausgegeben wird. Dieser wird mittels des Codes im Ausgabebereich erzeugt und wird verwendet, um in DiMo Tool Web das Modell zu veranschaulichen. Das Resultat hiervon ist in Abbildung 1 auf der rechten Seite zu sehen. Nach der Ausgabe ist die erste Iteration mit $i = 0$ abgeschlossen und i wird um eins hochgezählt. Daraufhin wird für $i = 1$ erneut $f_{D_{\mathtt{n}}}(i)$ berechnet. Da $D_{\mathtt{n}}$ nur einen Wert enthält, ist nun $f_{D_{\mathtt{n}}}(i) = \mathtt{None}$. Damit wird in DiMo signalisiert, dass alle Werte aus $D_{\mathtt{n}}$ bereits aufgezählt wurden und das Programm beendet werden kann.
In DiMo ist es auch möglich mehrere Parameter mit verschiedenen Definitionsbereichen in `PARAMETERS` kommagetrennt anzugeben. In diesem Fall wird bei der Ausführung das Kreuzprodukt der Definitionsbereiche der Parameter aufgezählt und die Formeln werden entsprechend instanziiert.

```
1  done.
2  Opening input file ‘./examples/version 0.3/binaryCode.dm´  done.
3  Constructing engine for formula scheme .....
4    Code(n)
5  Running engine ................................
6  
7  Running iteration 0
8    Constructing parameter evaluation ............ n=10
9    Instantiation yields formula ................. tt & -B(0) & B(1) & tt & tt & -B(2) & B(3) & tt & tt & -B(4)
10   Tidying yields ............................... -B(4) & B(3) & -B(2) & B(1) & -B(0)
11   Collecting propositions ...................... B(0), B(1), B(2), B(3), B(4)
12   Transformation to CNF yields ................. { { -B(4) } , { B(3) } , { -B(2) } , { B(1) } , { -B(0) } }
13   Getting new solver ........................... done.
14   Adding clauses ............................... done.
15   Solving ...................................... satisfiable!
16   Collecting relevant literals from solution ... done.
17 Instance n=10 .................................. satisfiable.
18   Satisfying assignment: -B(0), B(1), -B(2), B(3), -B(4)
19 <p>The binary representation of the decimal number <b>10 </b> is: </p><br><table><tr><th>0</th><th>1</th><th
       >0</th><th>1</th><th>0</th></tr><tr><th class='binary'>2<sup>4</sup></th> <th class='binary'>2<sup>3</
       sup></th> <th class='binary'>2<sup>2</sup></th> <th class='binary'>2<sup>1</sup></th> <th class='binary
       '>2<sup>0</sup></th> </tr></table>
20 Running iteration 1
21   Constructing parameter evaluation ............
22 stopped.
```

Listing 3: Ausgabe der Ausführung von DiMo auf dem Beispiel-Code

3 Erweiterung der DiMo-Sprache

Im Folgenden wird beschrieben, welche Schritte unternommen werden müssen, um die in 1.1 festgelegten Anforderungen zu implementieren. Dabei wird zuerst darauf eingegangen, welche syntaktischen und semantischen Erweiterungen für die Implementierung der Wörter vorgenommen wurden und danach darauf, wie diese umgesetzt wurden.

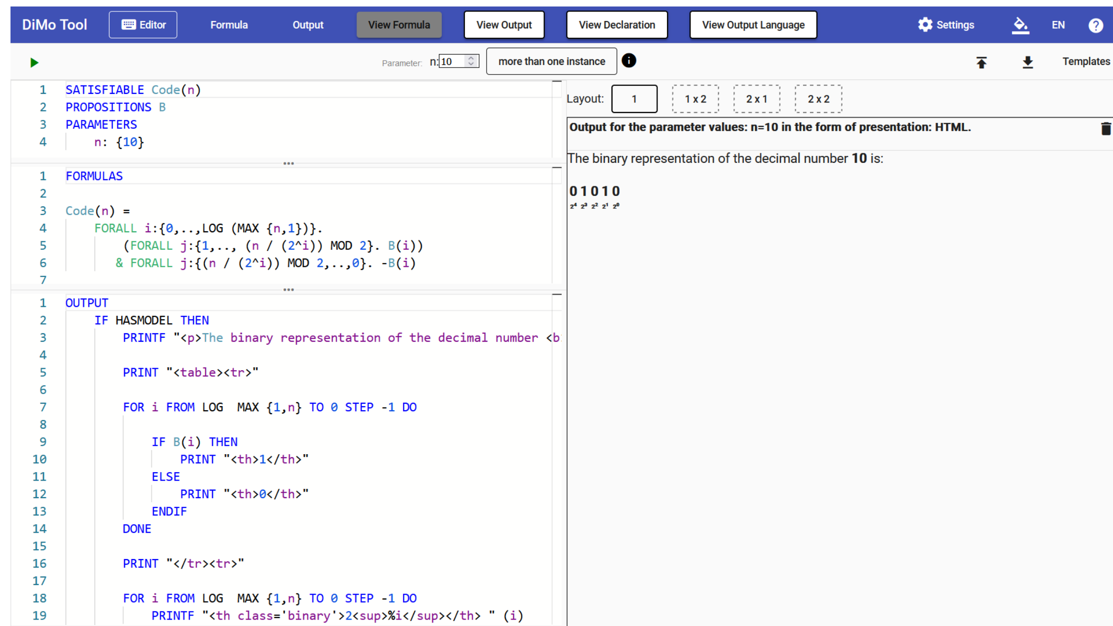

Abbildung 1: Benutzeroberfläche von DiMo Tool Web

3.1 Syntaktische und semantische Erweiterungen

In diesem Abschnitt wird beschrieben, wie die Anforderungen umgesetzt werden sollen.

Erweiterung im Header

In Listing 4 ist zu sehen, wie der neue Datentyp für Wörter in DiMo genutzt werden kann. Wortkonstanten können als Aneinanderreihung von Zeichen in Anführungszeichen angegeben werden, genauso wie es in vielen Programmiersprachen für Zeichenketten üblich ist. Z. B. steht `"abc"` in Zeile 3 für das Wort $w = abc$, das leere Wort ε kann mit der leeren Zeichenkette `""` beschrieben werden. Definitionsbereiche für Wortparameter können, wie bei Zahlen, explizit angegeben werden, indem alle Elemente einzeln aufgezählt werden (s. Zeile 3). Außerdem können reguläre Sprachen als Definitionsbereich festgelegt werden, indem ein regulärer Ausdruck angegeben wird (s. Zeile 4). Reguläre Ausdrücke sind so aufgebaut, wie in Definition 16 beschrieben, wobei die Konkatenation zweier Ausdrücke mit einem Unterstrich `_` und die Vereinigung, wie bei den Mengenoperationen von DiMo, mit `|` anstelle von `+` beschrieben wird. So kann das Symbol `+` verwendet werden, um die positive Hülle einer regulären Sprache zu beschreiben. Zudem gibt es, wie bei den Mengenoperationen, zusätzlich die Operatoren `&` und `-`, um den Schnitt und die Differenz von regulären Sprachen zu beschreiben und den Operator `^`, um Potenzen von Sprachen auszudrücken. Das Schlüsselwort `SIGMA` ist ein regulärer Ausdruck für ein beliebiges Zeichen aus dem Alphabet, ist also (für $\Sigma = \{a, b\}$) äquivalent zu dem Ausdruck `("a" | "b")`. Die Konkatenation von mehreren konstanten Zeichen kann auch als einzelne Zeichenkette zusammengefasst werden. Z. B. ist `"abc"_SIGMA*` äquivalent zu `"a"_"b"_"c"_SIGMA*`. Mit dem Schlüsselwort `COMPOSED` wird angegeben, auf welche Weise die Sprache des regulären Ausdrucks aufgezählt werden soll. Näheres hierzu wird in 3.1 beschrieben.
Das Alphabet kann sowohl implizit ermittelt werden, als auch explizit bei den Parametern mit angegeben werden. Eine explizite Angabe kann im Parameterabschnitt in Mengenschreibweise erfolgen (s. Zeile 5). Kommen innerhalb der Definitionsbereiche weitere Zeichen vor, die nicht im explizit angegebenen Alphabet enthalten sind, wird das

```
1 SATISFIABLE IsSubword(u, v)
2 PROPOSITIONS S
3 PARAMETERS u: {"ab", "abc"},
4            v: COMPOSED "["_("" | (("'"_("a" | "b"_("a" | "b")*)_"'")_(",
    '"_("a" | "b"_("a" | "b")*)_"'")*))_"]",
5            SIGMA: {" ", "'", ",", "a", "b", "[", "]"}
```

Listing 4: Header eines DiMo-Programms mit Wörtern

```
1   MapPropositionsU(u) = FORALL i: {1,..,|u|}. U(i, u(i)) & FORALL s:
    SIGMA-{u(i)}. -U(i, s)
```

Listing 5: Formelbereich eines DiMo-Programms mit Wörtern

Programm mit einer Fehlermeldung beendet.
Wird das Alphabet nicht explizit angegeben, besteht das implizit ermittelte Alphabet aus allen Zeichen, die in Wortkonstanten innerhalb der Definitionsbereiche der (Wort-)Parameter vorkommen. In diesem Beispiel könnte das Alphabet auch implizit ermittelt werden und würde in dem Fall aus denselben Zeichen bestehen, die in Zeile 5 explizit angegeben sind. Sind keine Zeichen in Wortkonstanten enthalten (z. B. in dem Ausdruck `"" | SIGMA*`), ist keine implizite Ermittlung möglich und das Programm wird mit einer Fehlermeldung beendet.

Erweiterung im Formelbereich

In Listing 5 ist zu sehen, wie die Syntax für Wortoperationen in Formeln ist. Hier ist zu sehen, dass das Schlüsselwort `SIGMA` eine Iterationsmenge für verallgemeinerte Junktoren darstellt. Die so dargestellte Menge ist die Menge an Zeichen, die (implizit oder explizit) als Alphabet festgelegt wurde. Hierbei haben Zeichen keinen eigenen Datentyp, sondern werden als Wort mit der Länge 1 behandelt. Zudem ist es auch möglich, (endliche) Wortmengen als Iterationsmenge bei verallgemeinerten Junktoren anzugeben. Diese sollen, wie bei den Definitionsbereichen, sowohl explizit angegeben werden können, als auch über reguläre Ausdrücke beschrieben werden können. Falls der reguläre Ausdruck eine unendliche Sprache beschreibt, soll das Programm mit einer Fehlermeldung beendet werden. Die Länge eines Wortes und das Zeichen an einer bestimmten Stelle eines Wortes können mit der Schreibweise aus Definition 7 ausgedrückt werden. Teilwörter können mit der Schreibweise aus Definition 8 beschrieben werden. Somit steht `|u|` für die Länge des Wortes `u`, `u(i)` für das `i`-te Zeichen in `u` (auch hier dargestellt als Wort der Länge 1) und `u(i, j)` für das Teilwort von `u`, welches an der Position `i` beginnt und an der Position `j` endet. Mit `u(i, ..)` bzw. `u(.., j)` enden die Teilwörter mit dem Ende des Wortes bzw. beginnen mit seinem Anfang. Die Konkatenation von Wörtern kann wie die Konkatenation von regulären Ausdrücken mit einem Unterstrich `_` als Operator ausgedrückt werden. Potenzen können wie Potenzen von Zahlen mit dem Operator `^` geschrieben werden. `u_v` steht also für die Konkatenation der Wörter `u` und `v`, und `u^3` für die 3-te Potenz von `u` (äquivalent zu `u_u_u`).

Erweiterung der Formelauswertung

Für die neu hinzugefügten Definitionsbereiche muss nun noch festgelegt werden, wie diese aufgezählt werden. Bei explizit aufgezählten Wortmengen werden die Elemente in

der Reihenfolge, in der sie angegeben wurden, aufgezählt. Für reguläre Ausdrücke wurden verschiedene Ansätze ausprobiert:

- Naive Aufzählung: Hier erfolgt die Aufzählung, indem für jedes Wort überprüft wird, ob es Teil der beschriebenen Sprache ist. Dieser Ansatz ist sehr ineffizient und führt bei der Verwendung endlicher Sprachen dazu, dass das Programm nicht terminiert.
- Zusammengesetzte Aufzählung: Bei diesem Ansatz wurde versucht, das Problem der naiven Aufzählung, dass viele Wörter generiert werden, die nicht gebraucht werden, zu beheben. Ein Nachteil der resultieren Implementierung ist, dass die Aufzählung im Allgemeinen nicht injektiv auf der aufzuzählenden Menge ist.
- Direkte Aufzählung: Hier wurde die direkte Aufzählung aus [8] implementiert, um eine auf $L\alpha$ injektive Aufzählung zu erhalten.

Sowohl die naive Aufzählung als auch die zusammengesetzte sind eher als prototypische Aufzählungen zu betrachten und nicht für den praktischen Gebrauch bestimmt. Bei Angabe des Definitionsbereiches kann vor dem regulären Ausdruck mit einem Schlüsselwort angegeben werden, auf welche Art die Sprache aufgezählt werden soll. Dabei steht das Schlüsselwort `NAIVE` für die naive, `COMPOSED` für die zusammengesetzte und `DIRECT` für die direkte Aufzählung. In den folgenden detaillierteren Beschreibungen der Aufzählungsarten sei...

- $\alpha \in$ RegEx der reguläre Ausdruck, der die aufzuzählende Sprache beschreibt
- Σ das Alphabet und
- $f_{\mathrm{M},\alpha}\colon \mathbb{N} \to L(\alpha) \cup \{\texttt{None}\}$ die zu berechnende Aufzählung von $L(\alpha)$, wobei „M“ hier ein Platzhalter für die verwendete Methode ist.

Naive Aufzählung Sei $f_{\Sigma^*}\colon \mathbb{N} \to \Sigma^*$ eine Aufzählung von Σ^*, die die Wörter der Länge nach aufzählt. Es gilt also $|f_{\Sigma^*}(i)| \leq |f_{\Sigma^*}(i+1)|$ für alle $i \in \mathbb{N}$. Zur Berechnung von $f_{\mathrm{naiv},\alpha}(i)$ werden nacheinander mithilfe von f_{Σ^*} die Wörter von Σ^* aufgezählt. Für jedes Wort $w \in \Sigma^*$ soll überprüft werden, ob $w \in L(\alpha)$ gilt. Die Wörter, für die dies zutrifft, werden gezählt. Ist w das $i+1$-te Wort, wird w zurückgegeben, ansonsten wird Σ^* weiter aufgezählt. Algorithmus 1 stellt das Verfahren in Pseudocode dar.

```
1  def f_Naiv(α, i):
2      j = 0
3      k = 0
4      while True:
5          w = f_Σ*(k)
6          while w ∉ L(α):
7              k += 1
8              w = f_Σ*(k)
9          if j == i:
10             return w
11         j += 1
12         k += 1
```

Algorithmus 1 : Naive Aufzählung von Sprachen regulärer Ausdrücke

Zu beachten ist, dass $f_{\text{naiv},\alpha}$ in dieser Form nicht total ist, falls $L = L(\alpha)$ endlich ist. Für $i \geq |L|$ werden nämlich alle $|L|$ Wörter in L aufgezählt, bis $j = |L| - 1$ gilt. Da die Bedingung in Zeile 9 dann noch nicht erfüllt ist, wird die äußere Schleife erneut durchlaufen. Nun wurden aber alle Wörter aus L schon aufgezählt, sodass alle weiter aufgezählten Wörter $w \in \Sigma^*$ nicht in L enthalten sind, sodass die Bedingung der inneren Schleife (Zeile 6) immer erfüllt bleibt. Somit bricht die innere Schleife dann niemals ab und das Programm terminiert nicht.

Ein weiteres Problem der naiven Aufzählung ist, dass sie sehr ineffizient ist:

Beispiel 3. Sei $\alpha = a \cdot b^*$ ein regulärer Ausdruck über $\Sigma = \{a, b\}$. Nach Definition 17 folgt $L(\alpha) = \{ab^n \mid n \in \mathbb{N}\}$. Somit gibt es für alle $n \in \mathbb{N}_{\geq 1}$ mit $w = ab^{n-1}$ nur ein $w \in L(\alpha)$ mit $|w| = n$. Da f_{Σ^*} die Wörter aus Σ^* der Länge nach aufzählt, muss das i-te von $f_{\text{naiv},\alpha}$ aufgezählte Wort eine Länge von $i + 1$ haben. Also müssen alle 2^i Wörter der Länge i vorher durch f_{Σ^*} aufgezählt werden, wodurch die Laufzeit des Algorithmus in diesem Fall in $\Omega(2^i)$ liegt.

Zusammengesetzte Aufzählung Bei der zusammengesetzten Aufzählung wird α rekursiv aufgezählt. Für die Berechnung wird zusätzlich eine grobe (möglicherweise unendliche) Schätzung $|\alpha|_\sim$ für $|L(\alpha)|$ genutzt, welche folgendermaßen definiert ist:

Definition 19. Sei $\alpha \in \text{RegEx}$. In den Basisfällen sei:

(1) $|\varepsilon|_\sim = 1$

(2) $|a|_\sim = 1$ für $a \in \Sigma$

Ist α aus $\alpha_1, \alpha_2 \in \text{RegEx}$ zusammengesetzt, werde $|\alpha|_\sim$ induktiv definiert:

(3) Fall $\alpha = \alpha_1 + \alpha_2$:
$$|\alpha|_\sim = |\alpha_1|_\sim + |\alpha_2|_\sim$$

(4) Fall $\alpha = \alpha_1 \cdot \alpha_2$:
$$|\alpha|_\sim = \begin{cases} |\alpha_1|_\sim \cdot |\alpha_2|_\sim, & \text{falls } \alpha_1 < \infty \text{ und } \alpha_2 < \infty \\ \infty, & \text{sonst} \end{cases}$$

(5) Fall $\alpha = \alpha_1^*$:
$$|\alpha|_\sim = \infty$$

Aufbauend auf dieser Schätzung wird nun $f_{\text{zsmges},\alpha}$ definiert. In den Basisfällen, falls $\alpha = w$ für $w \in \{\varepsilon\} \cup \Sigma$, folgt die Definition von $f_{\text{zsmges},\alpha}$ direkt aus Definition 17:

$$f_{\text{zsmges},\alpha}(i) = \begin{cases} w, & \text{falls } i = 0 \\ \texttt{None}, & \text{sonst} \end{cases}$$

In den induktiven Fällen seien nun $\alpha_1, \alpha_2 \in \text{RegEx}$. Ist α die Vereinigung $\alpha = \alpha_1 + \alpha_2$, gelte o. B. d. A. $|\alpha_1|_\sim \leq |\alpha_2|_\sim$. Dabei wird abwechselnd aus α_1 und α_2 aufgezählt. Ist $|\alpha_1|_\sim$ endlich und wurden alle Wörter aus $L(\alpha_1)$ aufgezählt, werden die restlichen Wörter aus $L(\alpha_2)$ aufgezählt. Formal ausgedrückt bedeutet das:

$$f_{\text{zsmges},\alpha}(i) = \begin{cases} f_{\text{zsmges},\alpha_1}\left(\left\lfloor \frac{i}{2} \right\rfloor\right), & \text{falls } i \text{ gerade und } \left\lfloor \frac{i}{2} \right\rfloor < |\alpha_1|_\sim \\ f_{\text{zsmges},\alpha_2}\left(\left\lfloor \frac{i}{2} \right\rfloor\right), & \text{falls } i \text{ ungerade und } \left\lfloor \frac{i}{2} \right\rfloor < |\alpha_1|_\sim \\ f_{\text{zsmges},\alpha_2}(i - |\alpha_1|_\sim), & \text{sonst} \end{cases}$$

Beschreibt $\alpha = \alpha_1 \cdot \alpha_2$ eine Konkatenation, werden alle Paare von Indizes $(i_1, i_2) \in I_1 \times I_2$ aufgezählt.

Definition 20. Für zwei reguläre Ausdrücke $\alpha_1, \alpha_2 \in \text{RegEx}$ seien die **Indexmengen** I_1, I_2 folgendermaßen definiert:

$$I_j = \begin{cases} \{0, \dots, |\alpha_j|_\sim - 1\}, & \text{falls } |\alpha_j|_\sim \text{ endlich} \\ \mathbb{N}, & \text{sonst} \end{cases}$$

für $j = 1, 2$.

Hiermit lässt sich $f_{\text{zsmges},\alpha}$ definieren als

$$f_{\text{zsmges},\alpha}(i) = \begin{cases} f_{\text{zsmges},\alpha_1}(i_1) \cdot f_{\text{zsmges},\alpha_2}(i_2), & \text{falls } (i_1, i_2) = f_{I_1 \times I_2}(i) \neq \texttt{None} \\ \texttt{None}, & \text{sonst} \end{cases}$$

für eine Aufzählung $f_{I_1 \times I_2}$ von $I_1 \times I_2$ nach Definition 18, die injektiv auf $I_1 \times I_2$ ist. Für alle $(i_1, i_2) \in I_1 \times I_2$ gibt es also nur ein $i \in \mathbb{N}$ mit $f_{I_1 \times I_2}(i) = (i_1, i_2)$. Außerdem wird vorausgesetzt, dass $i_1 \leq i$ und $i_2 \leq i$ für alle $i \in \mathbb{N}$ und $(i_1, i_2) = f_{I_1 \times I_2}(i)$ gilt. Eine Aufzählung $f_{I_1 \times I_2}$ mit diesen Eigenschaften war bereits vor der Erweiterung in DiMo implementiert, da bei Formeln mit mehreren Parametern auch das Kreuzprodukt der Definitionsbereiche aufgezählt wurde.

Die kleenesche Hülle $\alpha = \alpha_1^*$ wird mithilfe von Satz 14 rekursiv aufgezählt, indem ε für $i = 0$ zurückgegeben wird und $L(\alpha_1 \cdot \alpha_1^*)$ für $i > 0$ aufgezählt wird. Es gilt dann also

$$f_{\text{zsmges},\alpha}(i) = \begin{cases} \varepsilon, & \text{falls } i = 0 \\ f_{\text{zsmges},\alpha_1 \cdot \alpha_1^*}(i - 1), & \text{sonst} \end{cases}$$

Zu beachten ist, dass die zusammengesetzte Aufzählung nicht notwendigerweise injektiv auf $L(\alpha)$ ist. Es kann also vorkommen, dass manche Wörter mehrfach aufgezählt werden. Für $\alpha = \alpha_1 + \alpha_2$ mit $\alpha_1 = \alpha_2 = a$ für $a \in \Sigma$ ergibt sich beispielweise $f_{\text{zsmges},\alpha}(0) = f_{\text{zsmges},\alpha}(1) = a$. Zur Veranschaulichung der Funktionsweise dieser Aufzählung folgt nun noch ein Beispiel:

Beispiel 4. Sei $\alpha = a \cdot b^*$ wieder ein regulärer Ausdruck über $\Sigma = \{a, b\}$. Da α die Konkatenation von $\alpha_1 = a$ und $\alpha_2 = b^*$ ist, sind nach Definition 20 folgende Indexmengen definiert:

$$I_1 = \{0, \dots, |\alpha_1|_\sim - 1\} = \{0\} \quad \text{da } |\alpha_1|_\sim = |a|_\sim = 1 \text{ endlich}$$

sowie

$$I_2 = \mathbb{N} \quad \text{da } |\alpha_2|_\sim = |b^*|_\sim = \infty \text{ nicht endlich}$$

Da I_1 nur die Zahl 0 und I_2 alle natürlichen Zahlen enthält, folgt für die Aufzählung $f_{I_1 \times I_2}$ von $I_1 \times I_2$:

$$f_{I_1 \times I_2}(i) = (0, i) \quad \text{für alle } i \in \mathbb{N}$$

Für die zusammengesetzte Aufzählung $f_{\text{zsmges},\alpha}$ von $L(\alpha)$ gilt nun:

$$\begin{aligned} f_{\text{zsmges},\alpha}(i) &= \begin{cases} f_{\text{zsmges},\alpha_1}(i_1) \cdot f_{\text{zsmges},\alpha_2}(i_2), & \text{falls } (i_1, i_2) = f_{I_1 \times I_2}(i) \neq \texttt{None} \\ \texttt{None}, & \text{sonst} \end{cases} \\ &= f_{\text{zsmges},a}(0) \cdot f_{\text{zsmges},b^*}(i) \\ &= a \cdot f_{\text{zsmges},b^*}(i) \end{aligned}$$

Um $f_{\text{zsmges},\alpha}(i)$ zu berechnen, muss also a mit $f_{\text{zsmges},b^*}(i)$ konkateniert werden. Für die Berechnung von f_{zsmges,b^*} ergibt sich:

$$f_{\text{zsmges},b^*}(i) = \begin{cases} \varepsilon, & \text{falls } i = 0 \\ f_{\text{zsmges},b \cdot b^*}(i-1), & \text{sonst} \end{cases}$$

Hierbei ist $b \cdot b^*$ nun wieder eine Konkatenation mit den Indexmengen:

$$I'_1 = \{0, \ldots, |b|_\sim - 1\} = \{0\} \quad \text{da } |b|_\sim = 1 \text{ endlich}$$

und

$$I'_2 = \mathbb{N} \quad \text{da } |b^*|_\sim = \infty \text{ nicht endlich}$$

Somit gilt analog zu $f_{I_1 \times I_2}$ für $f_{I'_1 \times I'_2}$

$$f_{I'_1 \times I'_2}(i) = (0, i) \quad \text{für alle } i \in \mathbb{N}$$

und für $f_{\text{zsmges},b \cdot b^*}$:

$$\begin{aligned} f_{\text{zsmges},b \cdot b^*}(i) &= \begin{cases} f_{\text{zsmges},b}(i_1) \cdot f_{\text{zsmges},b^*}(i_2), & \text{falls } (i_1, i_2) = f_{I'_1 \times I'_2}(i) \neq \texttt{None} \\ \texttt{None}, & \text{sonst} \end{cases} \\ &= f_{\text{zsmges},b}(0) \cdot f_{\text{zsmges},b^*}(i) \\ &= b \cdot f_{\text{zsmges},b^*}(i) \end{aligned}$$

Nun kann man dieses Ergebnis bei der Gleichung für f_{zsmges,b^*} einsetzen:

$$\begin{aligned} f_{\text{zsmges},b^*}(i) &= \begin{cases} \varepsilon, & \text{falls } i = 0 \\ f_{\text{zsmges},b \cdot b^*}(i-1), & \text{sonst} \end{cases} \\ &= \begin{cases} \varepsilon, & \text{falls } i = 0 \\ b \cdot f_{\text{zsmges},b^*}(i-1), & \text{sonst} \end{cases} \end{aligned}$$

An dieser rekursiven Gleichung kann man leicht erkennen, dass $f_{\text{zsmges},b^*}(i) = b^i$ gelten muss. Für $f_{\text{zsmges},\alpha}$ ergibt sich damit:

$$\begin{aligned} f_{\text{zsmges},\alpha}(i) &= a \cdot f_{\text{zsmges},b^*}(i) \\ &= a \cdot b^i \end{aligned}$$

An diesem Beispiel lässt sich erkennen, dass nur Wörter generiert werden, deren Präfix a ist. Anders als bei der naiven Aufzählung werden also nicht alle Wörter bis zu einer bestimmten Länge generiert, wodurch die zusammengesetzte Aufzählung wesentlich schneller ist.

Nun muss noch gezeigt werden, dass das Verfahren trotz der Schätzung der Kardinalität korrekt ist, also dass folgender Satz gilt:

Satz 21. *Für alle regulären Ausdrücke* $\alpha \in \text{RegEx}$ *ist* $f_{\text{zsmges},\alpha}$ *eine Aufzählung nach Definition 18.*

Beweis. Satz 21 wird im Folgenden per Induktion über α gezeigt. Dafür muss für alle $\alpha \in \text{RegEx}$ gezeigt werden, dass $f_{\text{zsmges},\alpha}$ eine Aufzählung nach Definition 18 ist. Aus dieser Definition folgen diese drei Bedingungen, die gezeigt werden müssen:

(i) $f_{\text{zsmges},\alpha}(i) \in L(\alpha) \cup \{\texttt{None}\}$ für alle $i \in \mathbb{N}$

(ii) Für alle $w \in L(\alpha)$ existiert ein $i \in \mathbb{N}$ mit $f_{\text{zsmges},\alpha}(i) = w$.

(iii) $f_{\text{zsmges},\alpha}(i) = \texttt{None} \Rightarrow f_{\text{zsmges},\alpha}(i+1) = \texttt{None}$ für alle $i \in \mathbb{N}$

Außerdem muss folgende Eigenschaft der Abschätzung $|\alpha|_\sim$, die für die Korrektheit notwendig ist, gezeigt werden:

(iv) $f_{\text{zsmges},\alpha}(i) \neq \texttt{None}$ g. d. w. $i < |\alpha|_\sim$ für alle $i \in \mathbb{N}$

Sei $\alpha \in \text{RegEx}$.

Induktionsanfang: Im Basisfall folgt wegen $f_{\text{zsmges},\alpha}(0) = w \neq \texttt{None}$ und $f_{\text{zsmges},\alpha}(i) = \texttt{None}$ für $i \neq 0$ direkt (iv). Offensichtlich gelten (i), (ii) und (iii), sodass $f_{\text{zsmges},\alpha}$ eine Aufzählung von $L(\alpha)$ ist.

Induktionsvoraussetzung: Für α_1 und α_2 seien $f_{\text{zsmges},\alpha_1}$ und $f_{\text{zsmges},\alpha_2}$ Aufzählungen von $L(\alpha_1)$ bzw. $L(\alpha_2)$. Es gelten also (i), (ii) und (iii) für α_1 und α_2. Zudem gelte (iv) für α_1 und α_2.

Induktionsschritt:
Fall $\alpha = \alpha_1 + \alpha_2$:
Beweis für (iv): Sei $i \in \mathbb{N}$. Nach Induktionsvoraussetzung gilt im Fall $\left\lfloor \frac{i}{2} \right\rfloor < |\alpha_1|_\sim$ auf jeden Fall $f_{\text{zsmges},\alpha_1}\left(\left\lfloor \frac{i}{2} \right\rfloor\right) \neq \texttt{None}$. Da $|\alpha_1|_\sim \leq |\alpha_2|_\sim$ vorausgesetzt wurde, gilt somit auch $\left\lfloor \frac{i}{2} \right\rfloor < |\alpha_2|_\sim$ und $f_{\text{zsmges},\alpha_2}\left(\left\lfloor \frac{i}{2} \right\rfloor\right) \neq \texttt{None}$. Gilt $i < |\alpha|_\sim = |\alpha_1|_\sim + |\alpha_2|_\sim$, so gilt auch $i - |\alpha_1|_\sim < |\alpha_2|_\sim$, sodass für $\left\lfloor \frac{i}{2} \right\rfloor \geq |\alpha_1|_\sim$ auch $f_{\text{zsmges},\alpha_2}(i - |\alpha_1|_\sim) \neq \texttt{None}$ gilt. Für $i \geq |\alpha|_\sim$ gilt dann auch $i - |\alpha_1|_\sim \geq |\alpha_2|_\sim$, sodass per Induktion dann

$$f_{\text{zsmges},\alpha}(i) = f_{\text{zsmges},\alpha_2}(i - |\alpha_1|_\sim) = \texttt{None}$$

gelten muss. In jedem Fall der Definition von $f_{\text{zsmges},\alpha}$ gilt $f_{\text{zsmges},\alpha}(i) \neq \texttt{None}$ genau dann, wenn $i < |\alpha|_\sim$ gilt, wodurch (iv) auch für α erfüllt ist.

Beweis für (i): Sei $i \in \mathbb{N}$. Falls $f_{\text{zsmges},\alpha}(i) \neq \texttt{None}$ gilt, folgt aus der Definition von $f_{\text{zsmges},\alpha}$ und der Induktionsvoraussetzung direkt, dass $f_{\text{zsmges},\alpha}(i)$ im ersten Fall der Definition nur ein Wert aus $L(\alpha_1)$ sein kann und im zweiten und dritten Fall nur ein Wert aus $L(\alpha_2)$. Somit gilt $f_{\text{zsmges},\alpha}(i) \in L(\alpha_1) \cup L(\alpha_2) \cup \{\texttt{None}\}$ und (i) ist gezeigt.

Beweis für (ii): Sei $w \in L(\alpha)$, sodass $w \in L(\alpha_1)$ oder $w \in L(\alpha_2)$ gilt. Gilt $w \in L(\alpha_1)$, gibt es durch die Induktionsvoraussetzung (ii) ein $j \in \mathbb{N}$ mit $f_{\text{zsmges},\alpha_2}(j) = w$. Wegen (iv) gilt außerdem $j < |\alpha_1|_\sim$. Mit $i = 2j$ ist i offensichtlich gerade und es gilt $\left\lfloor \frac{i}{2} \right\rfloor = j < |\alpha_1|_\sim$. Somit tritt der erste Fall der Definition von $f_{\text{zsmges},\alpha}$ ein und es gilt $f_{\text{zsmges},\alpha}(i) = w$, wodurch (ii) für den Fall $w \in L(\alpha_1)$ gezeigt ist. Im Fall $w \in L(\alpha_2)$ gibt es ein $j \in \mathbb{N}$ mit $f_{\text{zsmges},\alpha_2}(j) = w$ und $j < |\alpha_2|_\sim$. Für den Fall, dass auch $j < |\alpha_1|_\sim$ gilt, sei $i = 2j + 1$. Offensichtlich ist i dann ungerade und es gilt $\left\lfloor \frac{i}{2} \right\rfloor = j < |\alpha_1|_\sim$. Somit tritt der zweite Fall der Definition von $f_{\text{zsmges},\alpha}$ ein und es gilt $f_{\text{zsmges},\alpha}(i) = w$. Im Fall $j \geq |\alpha_1|_\sim$ sei $i = j + |\alpha_1|_\sim$. Dann tritt der dritte Fall der Definition ein und auch dann gilt:

$$f_{\text{zsmges},\alpha}(i) = f_{\text{zsmges},\alpha_2}(j + |\alpha_1|_\sim - |\alpha_1|_\sim) = w$$

Somit ist (ii) auch für den Fall $w \in L(\alpha_1)$ gezeigt.

Beweis für (iii): Nach (iv) kann im ersten und zweiten Fall der Definition von $f_{\text{zsmges},\alpha}$ nicht `None` herauskommen. Im dritten Fall ist $f_{\text{zsmges},\alpha}(i)$ genau dann `None`, wenn

$$f_{\text{zsmges},\alpha_2}(i - |\alpha_1|_\sim) = \texttt{None}$$

gilt. Da (iii) induktiv für α_2 gilt, gilt dann auch

$$f_{\text{zsmges},\alpha}(i + 1) = f_{\text{zsmges},\alpha_2}(i - |\alpha_1|_\sim + 1) = \texttt{None},$$

sodass (iii) auch für α gilt.
Somit gelten alle drei Bedingungen (i), (ii) und (iii), wodurch gezeigt ist, dass $f_{\text{zsmges},\alpha}$ eine Aufzählung ist.

Fall $\alpha = \alpha_1 \cdot \alpha_2$:
Nach (iv) folgt für die Indexmengen I_1 und I_2 direkt:

Folgerung 22.

$$I_j = \{i \in \mathbb{N} \mid f_{\text{zsmges},\alpha_j}(i) \neq \texttt{None}\} \text{ \textit{für} } j = 1, 2$$

Beweis für (i): Sei $(i_1, i_2) \in I_1 \times I_2$ ein Paar von Indizes. Wegen Folgerung 22 gilt für $j = 1, 2$ dann $f_{\text{zsmges},\alpha_j}(i_j) \neq \texttt{None}$. Da nach Induktionsvoraussetzung (i) für α_1 und α_2 gilt, folgt dann, dass $f_{\text{zsmges},\alpha_j}(i_j) \in L(\alpha_j)$ für $j = 1, 2$ gelten muss. Werden nun $f_{\text{zsmges},\alpha_1}(i_1)$ und $f_{\text{zsmges},\alpha_2}(i_2)$ konkateniert, erhält man daher auch ein Wort aus $L(\alpha_1) \cdot L(\alpha_2)$. Hieraus folgt dann direkt (i) für α.

Beweis für (ii): Sei $w \in L(\alpha)$. Dann gibt es zwei Wörter $w_1 \in L(\alpha_1)$ und $w_2 \in L(\alpha_2)$, sodass $w = w_1 \cdot w_2$ gilt. Wegen (ii) gibt es auch zwei Indizes $i_1 \in I_1$ und $i_2 \in I_2$ mit $f_{\text{zsmges},\alpha_1}(i_1) = w_1$ sowie $f_{\text{zsmges},\alpha_2}(i_2) = w_2$. Da $f_{I_1 \times I_2}$ eine Aufzählung von $I_1 \times I_2$ ist, gibt es ein $i \in \mathbb{N}$, sodass $f_{I_1 \times I_2}(i) = (i_1, i_2)$ ergibt. Nach Definition von $f_{\text{zsmges},\alpha}$ gilt nun

$$(f_{\text{zsmges},\alpha}(i) = f_{\text{zsmges},\alpha_1}(i_1) \cdot f_{\text{zsmges},\alpha_2}(i_2) = w_1 \cdot w_2 = w,$$

womit (ii) für α gezeigt ist.

Beweis für (iii): Sei $i \in \mathbb{N}$ mit $f_{\text{zsmges},\alpha}(i) = \texttt{None}$. Dann muss nach Definition von $f_{\text{zsmges},\alpha}$ auch $f_{I_1 \times I_2}(i) = \texttt{None}$ gelten. Da $f_{I_1 \times I_2}$ auch eine Aufzählung nach Definition 18 ist, gilt für diese Funktion auch (iii), sodass aus $f_{I_1 \times I_2}(i) = \texttt{None}$ direkt $f_{I_1 \times I_2}(i + 1) = \texttt{None}$ folgt. Somit tritt für $f_{\text{zsmges},\alpha}(i + 1)$ auch der zweite Fall in der Definition von $f_{\text{zsmges},\alpha}$ ein und es gilt $f_{\text{zsmges},\alpha}(i + 1) = \texttt{None}$. Damit ist (iii) für α gezeigt.

Beweis für (iv): Da $f_{I_1 \times I_2}$ eine Aufzählung nach Definition 18 ist, gibt es für jedes Paar $(i_1, i_2) \in I_1 \times I_2$ mindestens ein $i \in \mathbb{N}$ mit $f_{I_1 \times I_2}(i) = (i_1, i_2)$. Also muss es mindestens $|I_1 \times I_2|$ natürliche Zahlen $i \in \mathbb{N}$ mit $f_{I_1 \times I_2}(i) \neq \texttt{None}$ geben. Damit das möglich ist, muss $f_{I_1 \times I_2}(i) \neq \texttt{None}$ für alle $i \in \mathbb{N}$ mit $i < |I_1 \times I_2|$ gelten, ansonsten wäre (iii) für $f_{I_1 \times I_2}$ verletzt. Aus Folgerung 22 folgt außerdem direkt $|I_j| = |\alpha_j|_\sim$ für $j = 1, 2$. Wegen

$$|I_1 \times I_2| = |I_1| \cdot |I_2| = |\alpha_1|_\sim \cdot |\alpha_2|_\sim = |\alpha|_\sim$$

gilt für ein $i < |\alpha|_\sim$ dann auch $i < |I_1 \times I_2|$. Wie oben gezeigt, muss dann $f_{I_1 \times I_2}(i) \neq \texttt{None}$ und somit auch $f_{\text{zsmges},\alpha}(i) \neq \texttt{None}$ gelten. Da $f_{I_1 \times I_2}$ injektiv auf $I_1 \times I_2$ ist, gibt es für alle Paare $(i_1, i_2) \in I_1 \times I_2$ höchstens ein $i \in \mathbb{N}$ mit $f_{I_1 \times I_2}(i) = (i_1, i_2)$. Somit gibt es höchstens $|I_1 \times I_2|$ Zahlen $i \in \mathbb{N}$ mit $f_{I_1 \times I_2} \in I_1 \times I_2$. Also muss für $i \geq |\alpha|_\sim = |I_1 \times I_2|$

daher $f_{I_1 \times I_2}(i) = \texttt{None}$ und somit auch $f_{\text{zsmges},\alpha}(i) = \texttt{None}$ gelten. Damit ist auch (iv) für α gezeigt.

Fall $\alpha = \alpha_1^*$:
Beweis für (iv): Da in dem Fall $|\alpha|_\sim = \infty$ festgelegt wurde, gilt $i < |\alpha|_\sim$ für alle $i \in \mathbb{N}$. Damit (iv) zutrifft, muss für alle $i \in \mathbb{N}$ also $f_{\text{zsmges},\alpha}(i) \neq \texttt{None}$ gelten. Dass dies für $i = 0$ zutrifft, folgt direkt aus der Definition von $f_{\text{zsmges},\alpha}$, da dann $f_{\text{zsmges},\alpha}(i) = \varepsilon$ ist. Für $i > 0$ wird auf die Aufzählung der Konkatenation von α_1 und α_1^* zurückgegriffen. Seien hierbei I_1 und I_2 die Indexmengen für α_1 bzw. α_1^* nach Definition 20. Da durch Definition 19 direkt $|\alpha_1|_\sim > 0$ folgt, muss $I_1 \neq \emptyset$ gelten. Da $|\alpha_1^*|_\sim = \infty$ festgelegt wurde, gilt dann $I_2 = \mathbb{N}$. Also ist I_2 unendlich und somit auch $I_1 \times I_2$, da I_1 nicht leer ist. Für eine Aufzählung $f_{I_1 \times I_2}$ von $I_1 \times I_2$ gilt dann auf jeden Fall $f_{I_1 \times I_2}(i-1) \neq \texttt{None}$ und nach Definition von $f_{\text{zsmges},\alpha_1 \cdot \alpha_1^*}$ dann auch $f_{\text{zsmges},\alpha_1 \cdot \alpha_1^*}(i-1) \neq \texttt{None}$. Damit ist (iv) für α gezeigt.

Beweis für (i) per Induktion über i:
Fall $i = 0$: Nach Definition gilt $f_{\text{zsmges},\alpha}(i) = \varepsilon \in L(\alpha)$. Damit ist der Basisfall für (i) direkt gezeigt.
Fall $i > 0$: Als Induktionsvoraussetzung werde $f_{\text{zsmges},\alpha}(i') \in L(\alpha)$ für alle $i' < i$ angenommen. Wegen $i > 0$ gilt:

$$(f_{\text{zsmges},\alpha}(i) = f_{\text{zsmges},\alpha_1 \cdot \alpha_1^*}(i-1)$$

Seien I_1 und I_2 wieder die Indexmengen für α_1 bzw. α_1^* nach Definition 20 und $f_{I_1 \times I_2}$ eine Aufzählung von $I_1 \times I_2$. Dann gilt:

$$(f_{\text{zsmges},\alpha}(i) = f_{\text{zsmges},\alpha_1}(i_1) \cdot f_{\text{zsmges},\alpha_1^*}(i_2) \text{ für } (i_1, i_2) = f_{I_1 \times I_2}(i-1)$$

Nach Induktionsvoraussetzung für (i) gilt $f_{\text{zsmges},\alpha_1}(i_1) \in L(\alpha_1)$. Da außerdem $i_2 \leq i - 1$ gilt, folgt $i_2 < i$. Somit ist die Induktionshypothese anwendbar und es folgt:

$$(f_{\text{zsmges},\alpha_1^*}(i_2) = f_{\text{zsmges},\alpha}(i_2) \in L(\alpha)$$

Da also sowohl $f_{\text{zsmges},\alpha_1}(i_1) \in L(\alpha_1)$ als auch $f_{\text{zsmges},\alpha_1^*}(i_2) \in L(\alpha_1^*)$ gilt, folgt nach Satz 14, dass $f_{\text{zsmges},\alpha_1}(i_1) \cdot f_{\text{zsmges},\alpha_1^*}(i_2) \in L(\alpha)$ gelten muss. Damit ist auch der induktive Fall und somit (i) für α gezeigt.

Beweis für (ii): Sei $w \in L(\alpha)$. Nun werde (ii) per Induktion über $|w|$ gezeigt.
Fall $|w| = 0$: Es gilt $w = \varepsilon$ und für $i = 0$ ergibt sich $f_{\text{zsmges},\alpha}(i) = w$. Damit ist der Basisfall für (ii) gezeigt.
Fall $|w| > 0$: Als Induktionshypothese werde angenommen, dass es für alle $w' \in L(\alpha)$ mit $|w'| < |w|$ ein $i \in \mathbb{N}$ gibt, sodass $f_{\text{zsmges},\alpha}(i) = w'$. Nach Satz 14 gibt es ein $w_1 \in L(\alpha_1)$ und $w_2 \in L(\alpha_1^*)$ mit $w = w_1 \cdot w_2$ und $w_1 \neq \varepsilon$. Hier darf $w_1 \neq \varepsilon$ angenommen werden, da nach Definition 13 jedes nicht-leere Wort aus $L(\alpha_1^*)$ ein Präfix aus $L(\alpha_1)$ haben muss. Durch Induktionshypothese für (ii) gibt es ein $i_1 \in \mathbb{N}$ mit $f_{\text{zsmges},\alpha_1}(i_1) = w_1$. Wegen $w_1 \neq \varepsilon$ gilt $|w_2| < |w|$, und nach Induktionshypothese muss es ein $i_2 \in \mathbb{N}$ mit $f_{\text{zsmges},\alpha}(i_2) = w_2$ geben. Seien nun I_1 und I_2 Indexmengen nach Definition 20 für α_1 bzw. α_1^* und $f_{I_1 \times I_2}$ eine Aufzählung von $I_1 \times I_2$. Offensichtlich gilt $i_1 \in I_1$ sowie $i_2 \in I_2$.

Also gibt es ein $i' \in \mathbb{N}$ mit $(i_1, i_2) = f_{I_1 \times I_2}(i')$. Mit $i = i' + 1$ gilt:

$$\begin{aligned} f_{\text{zsmges},\alpha}(i) &= f_{\text{zsmges},\alpha_1 \cdot \alpha_1^*}(i-1) \\ &= f_{\text{zsmges},\alpha_1 \cdot \alpha_1^*}(i') \\ &= f_{\text{zsmges},\alpha_1}(i_1) \cdot f_{\text{zsmges},\alpha_1^*}(i_2) \\ &= w_1 \cdot w_2 \\ &= w \end{aligned}$$

Somit ist (ii) auch für den induktiven Fall gezeigt.

Beweis für (iii): Im Beweis für (iv) wurde bereits argumentiert, dass $f_{\text{zsmges},\alpha}(i) \neq$ `None` für beliebige $i \in \mathbb{N}$ gilt. Damit ist (iii) trivialerweise für α erfüllt.

Durch Induktion folgt nun, dass $f_{\text{zsmges},\alpha}$ immer eine Aufzählung von $L(\alpha)$ ist.

□

Direkte Aufzählung Bei der direkten Aufzählung wurde die direkte Aufzählung aus [8] implementiert. Wie in dem Paper wurden hier Length-Ordered-Lists von Wörtern verwendet, damit die Aufzählung $f_{\text{direkt},\alpha}$ injektiv auf $L(\alpha)$ ist. Eine Length-Ordered-List ist eine (möglicherweise unendliche) verkettete Liste, deren Elemente aufsteigend nach der Länge sortiert sind. Im Folgenden wird ein Wort u kleiner als ein anderes Wort v bezeichnet, wenn u kürzer als v ist, oder wenn u und v gleich lang sind und u lexikographisch kleiner [9, S. 87] als v ist. Um $L(\alpha)$ aufzuzählen, wird aus α rekursiv eine Length-Ordered-List l generiert. Ist also $\alpha = a$ für $a \in \Sigma$, enthält l nur a als Element. Ist α hingegen aus den regulären Ausdrücken $\alpha_1, \alpha_2 \in$ RegEx zusammengesetzt, werden zuerst diese Unterausdrücke rekursiv in Length-Ordered-Lists l_1, l_2 umgewandelt und diese anschließend zu einer Liste zusammengeführt. Das Verfahren, wie diese Listen zusammengeführt werden, ist in [8] beschrieben. Um beispielsweise die Vereinigung zweier Listen zu berechnen, wird, falls eine Liste leer ist, die nicht-leere Liste zurückgegeben. Ansonsten werden die jeweils ersten Elemente der Listen miteinander verglichen, wobei das kleinere Element aus seiner Liste entfernt und an die Ergebnisliste angehängt wird. Sind die Elemente gleich, werden beide Elemente entfernt und nur ein Element an die Ergebnisliste angehängt, um Duplikate zu vermeiden [8, Absatz 3.1]. So wird die Injektivität auf $L(\alpha)$ von $f_{\text{direkt},\alpha}$ sichergestellt. Die Konkatenation zweier Listen ist mithilfe der Vereinigung implementiert [8, Absatz 3.2] und die kleenesche Hülle wiederum über die Konkatenation [8, Absatz 3.3].

3.2 Implementierung der Erweiterungen

Nun wird beschrieben, wie Erweiterungen der Datentypen in DiMo implementiert werden können. DiMo ist in der funktionalen Programmiersprache OCaml [10] geschrieben. Um die neue Syntax zu implementieren, müssen zuerst der Lexer und der Parser von DiMo angepasst werden. Diese sind mithilfe des Lexergenerators `ocamllex` und des Parsergenerators `ocamlyacc` implementiert. Im Folgenden werden die wichtigsten Schritte bei der Anpassung des Lexers und Parsers beschrieben. Eine ausführliche Dokumentation ist im Online-Handbuchs von OCaml [10, Kapitel 13] zu finden.

Erweiterung am Lexer

Der DiMo-Lexer ist in der Datei `lexer.mll` implementiert. Gibt es neue Schlüsselwörter oder Operatoren, werden diese hier als Token definiert. Dies ist das Schema zur Beschreibung von Token in dieser Datei:

```
rule token = parse
  | <regex-1>              { <token-1> }
  | <regex-2>              { <token-2> }
  | ...
  | <regex-n>              { <token-n> }
```

Die Tokenwerte in den geschweiften Klammern sind OCaml-Ausdrücke, die einen Wert für das Token beschreiben, der an den Parser weitergegeben wird. Der Typ des Tokens ist ein OCaml Variant-Typ, dessen Konstruktoren im Parser festgelegt werden. Um ein neues Token hinzuzufügen, muss also eine weitere Zeile zu der Regel `token` hinzugefügt werden. Z. B. ist zum Hinzufügen des Schlüsselworts `SIGMA` folgende Zeile erforderlich:

```
  | "SIGMA"                { TSIGMA }
```

`TSIGMA` ist hier ein Konstruktor ohne Parameter, der für das Schlüsselwort `SIGMA` steht. Eine Beschreibung eines Tokens kann auch Parameter enthalten, so sieht z. B. die Beschreibung für Int-Token, wie sie zur Darstellung von (natürlichen) Ganzzahlen verwendet werden, folgendermaßen aus:

```
  | ['0'-'9']+ as lxm      { TINT(int_of_string lxm) }
```

Ein komplexeres Beispiel von einem Token mit mehreren Konstruktorparametern ist das Token für Strings, welches vorher für Strings in der Ausgabesprache verwendet wurde und nun auch für Wortkonstanten verwendet wird. Die Implementierung dieses Tokens wurde von

```
  | "\"" [^ '\"']* "\""  as str   { TSTRING(str) }
```

zu

```
  | "\"" ([^ '\"'] | "\\\"")* "\""  as str
     { TSTRING(Scanf.unescaped (String.sub str 1 ((String.length str) - 2))) }
```

geändert. Die Änderung hat zum einen den Effekt, dass durch den Aufruf von `String.sub` die Anführungszeichen, in denen der String steht, nicht mitgespeichert werden. Zum anderen ist es durch die Anpassung des regulären Ausdrucks möglich, dass Zeichenketten mit Escape-Sequenzen als Token erkannt werden. Diese werden durch den Aufruf von `Scanf.unescaped` in das entsprechende Zeichen umgewandelt. Stehen zum Beispiel in der Eingabe die Zeichen `"a\nbc"`, wird `\n` in einen Zeilenvorschub (ASCII-Code 10 [11]) umgewandelt. Dadurch ist es sowohl möglich, Leerzeilen als Zeichen in Wörtern zu haben, als auch Leerzeilen im Ausgabeprogramm auszugeben. Mit `as lxm` wird das Lexem (also die Zeichenkette, die als Token erkannt wurde) in der Variable `lxm` gespeichert, auf welche dann im Code für den Tokenwert zugegriffen werden kann. In dem Beispiel ist durch den regulären Ausdruck garantiert, dass das Lexem eine Zahl im Dezimalsystem beschreibt, sodass es mit `int_of_string` in eine OCaml-Zahl umgewandelt werden kann. Diese wird nun an den Token-Konstruktor `TINT` übergeben.

Erweiterung am Parser

Der Parser für DiMo ist in der Datei `parser.mly` mittels einer Grammatik implementiert. Damit der erweiterte Lexer kompiliert, müssen dort verwendete Konstruktoren mit `%token` im Parser deklariert werden. Um z. B. den oben hinzugefügten Konstruktor `TSIGMA` zu deklarieren, muss folgende Zeile im Parser hinzugefügt werden:

```
%token TSIGMA
```

Bei der Deklaration von Konstruktoren mit Parametern wird der Typ der Parameter zwischen < und >-Symbole geschrieben. Somit sieht die Deklaration des Konstruktors `TINT`, der ein `int` als Parameter hat, folgendermaßen aus:

```
%token <int> TINT
```

Nun kann die Grammatik erweitert werden, indem bestehende Nichtterminale um neue Ableitungen ergänzt werden und ggf. neue Nichtterminale hinzugefügt werden. Die Regeln werden hier nach folgendem Schema beschrieben:

```
<nichtterminal>:
    <symbol1> TTOKEN <symbol3> {<AST-Knoten1>}
  | <symbol1>                  {<AST-Knoten2>}
```

Dabei steht `TTOKEN` für ein im Lexer definiertes Token, `<symbol1>` und `<symbol3>` für weitere Nichtterminale und `<AST-Knoten1>` und `<AST-Knoten2>` für OCaml-Ausdrücke, die jeweils den aus der Ableitung resultierenden Knoten im Syntaxbaum beschreiben. In diesem Ausdruck kann man mit den Variablen `$1`, `$2`, ... auf den jeweiligen Unterknoten, der durch das 1., 2., ... Symbol in der Ableitung beschrieben ist, zugreifen. Im obigen Schema würde innerhalb von `<AST-Knoten1>` z. B. `$2` für den vom Lexer zurückgegebenen Tokenwert für `TTOKEN` stehen und `$3` für den aus `<symbol3>` abgeleiteten Knotenwert. Die Typen der Knotenwerte sind in den Dateien `terms.ml` und `types.ml` definiert. Oft heißen die Typen genauso wie das Nichtterminal auf der linken Seite der Ableitungsregel und sind als OCaml Variant-Typ umgesetzt. In manchen Fällen ist der Typ auch zusammengesetzt aus selbst definierten Typen und eingebauten Typen von OCaml (z. B. Tupel oder Listen).

In Listing 6 sind einige Ableitungen des Nichtterminals `domain`, welches für Definitionsbereiche steht, zu sehen. Hier wurde die Syntax der Definitionsbereiche, die in 3.1 beschrieben wurde, implementiert. Die Syntax für endliche Wortmengen, bei der alle enthaltenen Wörter explizit aufgezählt werden, ist hier in Zeile 3 festgelegt, indem auf das Nichtterminal `wordconstants` abgeleitet wird, welches innerhalb von geschweiften Klammern (dargestellt durch die Token `TLBRACE` und `TRBRACE`) steht. In Zeile 4 und 5 wird festgelegt, dass der Definitionsbereich mit einem regulären Ausdruck beschrieben werden kann. Das Token `TRENUMKIND` steht für eines der Schlüsselwörter, mit denen angegeben werden kann, wie die Sprache aufgezählt werden soll (z. B. `COMPOSED`, s. 3.1). In Zeile 4 ist durch die Übergabe von `Regex.Naive` an den `RegularLanguage`-Konstruktor implementiert, dass standardmäßig naiv aufgezählt werden soll, wenn kein solches Schlüsselwort angegeben ist.

```
domain:
    TLBRACE intconstants TRBRACE    { FinSet($2) }
  | TLBRACE wordconstants TRBRACE   { FinWordSet($2) }
  | regex                           { RegularLanguage(Regex.Naive, $1) }
  | TRENUMKIND regex                { RegularLanguage($1, $2) }
```

Listing 6: Ausschnitt aus den Ableitungen des Nichtterminals `domain`

Der Knotentyp für diese Regel heißt auch `domain` und ist ein Variant-Typ mit den Konstruktoren `From` (für unendliche Bereiche von natürlichen Zahlen), `FromTo` (für endliche Bereiche von natürlichen Zahlen), `FinSet` (für explizit aufgezählte Bereiche von natürlichen Zahlen). Für Sprachen wurden zudem die Konstruktoren `FinWordSet` (für explizit aufgezählte Wortmengen) mit einer Liste von Strings als Parameter und `RegularLanguage` (für Sprachen, die durch einen regulären Ausdruck beschrieben sind) hinzugefügt. Das

Nichtterminal `wordconstants` für explizit aufgezählte Wortmengen wurde folgendermaßen definiert:

```
wordconstants:
      TSTRING                      { [ $1 ] }
    | TSTRING TCOMMA wordconstants { $1 :: $3 }
```

Wie oben beschrieben, steht `TSTRING` für eine Zeichenkette.
Für reguläre Ausdrücke wurde das neue Nichtterminal `regex` hinzugefügt:

```
regex:
      TSTRING                 { RegConstant($1) }
    | TSIGMA                  { RegAlphabet }
    | regex TAND regex
        { RegBinOp("&",$1,$3,(fun _ _ -> failwith "...")) }
    | regex TOR regex
        { RegBinOp("|",$1,$3,(fun r1 -> fun r2 -> Re.alt [r1;r2])) }
    | regex TUNDERSCORE regex
        { RegBinOp("_",$1,$3,(fun r1 -> fun r2 -> Re.seq [r1;r2])) }
    | regex TNEG regex
        { RegBinOp("-",$1,$3,(fun _ _ -> failwith "...")) }
    | regex TMULT             { RegUnOp("*",$1,Re.rep) }
    | regex TPLUS             { RegUnOp("+",$1,Re.rep1) }
    | regex TPOW term         { RegPower($1,$3) }
    | TLPAREN regex TRPAREN   { $2 }
```

Die Ableitungen des Nichtterminals `paramorconsts` wurden so ergänzt, dass Formeldefinitionen neben variablen Parametern auch konstante Parameter des neuen Datentyps haben können:

```
paramorconsts:
      TPARAM                       { [ Param($1) ] }
    | TINT                         { [ IntConst($1) ] }
    | TPARAM TCOMMA paramorconsts  { (Param($1)) :: $3 }
    | TINT TCOMMA paramorconsts    { (IntConst($1)) :: $3 }
    | TSTRING                      { [ WordConst($1) ] }
    | TSTRING TCOMMA paramorconsts { (WordConst($1)) :: $3 }
```

Auch hier werden die Konstanten wieder mit dem Token `TSTRING` umgesetzt.

Durch Ableitungen des Nichtterminals `symbset` wird die Syntax für Mengen definiert, die bei den verallgemeinerten Junktoren `FORALL` und `FORSOME` aufgezählt werden können. Damit auch hier durch reguläre Ausdrücke reguläre Sprachen angegeben werden können, wurde hier eine Ableitung zu der neu hinzugefügten Regel `regex` hinzugefügt.

```
symbset:
      TLBRACE terms TRBRACE                          { SmallSet($2) }
    | TLBRACE term TCOMMA term TCOMMA TDOTS TCOMMA term TRBRACE
        { Enumeration($2,$4,$8) }
    | TLBRACE term TCOMMA TDOTS TCOMMA term TRBRACE
        { Enumeration($2,
                      BinOp("+",$2,IntConst(1),int_term_bin_op (+)),
                      $6) }
    | symbset TOR symbset    { BinSetOp("|",$1,$3,ValueSet.union) }
    | symbset TAND symbset   { BinSetOp("&",$1,$3,ValueSet.inter) }
    | symbset TNEG symbset   { BinSetOp("\\",$1,$3,ValueSet.diff) }
    | TSIGMA                 { FiniteLanguage(RegAlphabet) }
    | TREG regex             { FiniteLanguage($2) }
```

Das Nichtterminal `term` steht für Terme, die in Parametern von Variablen bzw. Hilfsformeln oder in Mengen genutzt werden können. Für Wörter wurden hier außer Konstanten

(`TSTRING`) nach 3.1 auch die Konkatenation von Wörten mittels Unterstrich `_` (Zeile 16), Bestimmung der Länge von Wörtern mit Betragsstrichen (Zeile 21), der Zugriff auf einzelne Zeichen eines Wortes (Zeile 23) sowie die Extraktion von Teilwörten (Zeile 25) syntaktisch festgelegt.

```
term:
      TINT                      { IntConst($1) }
    | TSTRING                   { WordConst($1) }
    | TNEG term
        { BinOp("-",IntConst(0), $2,
                int_term_bin_op (fun x -> fun y -> x-y)) }
    | TPARAM                    { Param($1) }
    | term TBINOP term          { let (s,f) = $2 in BinOp(s,$1,$3,f) }
    | term TPOW term            { BinOp("^",$1,$3,power_operator) }
    | term TPLUS term           { BinOp("+",$1,$3, int_term_bin_op (+)) }
    | term TMULT term
        { BinOp("*",$1,$3,int_term_bin_op (fun x -> fun y -> x*y)) }
    | term TNEG term
        { BinOp("-",$1,$3,int_term_bin_op (fun x -> fun y -> x-y)) }
    | term TDIV term            { BinOp("/",$1,$3,int_term_bin_op (/)) }
    | term TUNDERSCORE term     { BinOp("_",$1,$3,word_concat) }
    | TUNOP term                { let (s,f) = $1 in UnOp(s,$2,f) }
    | TMIN symbset              {...}
    | TMAX symbset              {...}
    | TLPAREN term TRPAREN      { $2 }
    | TOR term TOR              { UnOp("|.|",$2,word_length_op) }
    | term TLPAREN term TRPAREN
        { BinOp("(.)",$1,$3,symbol_of_word_at_pos_op) }
    | term TLPAREN slice TRPAREN
        { WordSlice($1,$3) }
```

Implementierung des Datentyps

Vor dieser Arbeit war der Wert jedes Terms in DiMo eine Ganzzahl, wodurch keine Unterscheidung von verschiedenen Typen nötig war. So konnte jeder Wert eines Terms als Ganzzahl vom OCaml-Typ `int` behandelt werden. Durch das Hinzufügen von Wörtern muss DiMo jetzt intern zwischen Zahlen und Wörtern unterscheiden. Hierbei wurde sich dafür entschieden, vorerst auf die Implementierung eines Typecheckers zu verzichten und dies später vorzunehmen. Dadurch kommt es bei dem Versuch, Operationen auf einem Wert des falschen Datentyps auszuführen, zu Laufzeitfehlern (z. B. bei Addition einer Zahl mit einem Wort oder Konkatenation von Zahlen). In solchen Fällen wird das Programm mit einer Fehlermeldung beendet.
Bei der Implementierung des Datentyps wurde darauf aufgebaut, dass es bereits einen Variant-Typ `term_types` mit dem Konstruktor `Int` gab, der ein `int` als Parameter hat. Dieser Typ wurde bisher verwendet, um die Werte von Parametern in einer Map zu speichern. Nun wurde zu diesem Typ noch ein weiterer Konstruktor `Word` für Terme vom DiMo-Typ Wort hinzugefügt:

```
type term_types = Int of int
                | Word of string
```

Einige Stellen im Code mussten noch angepasst werden, da dort Werte von Termen als `int` behandelt wurden.

Änderung von Knotentypen Einige Typen von AST-Knoten mussten angepasst werden. Die Unterschiede des Typen von AST-Knoten des Nichtterminals `term` sind in

Listing 7 und Listing 8 zu sehen. Dort ist zu sehen, dass der Typ von AST-Knoten des Nichtterminals `term` von `intTerm` in `term` umbenannt wurde. Auch die Konstruktoren dieses Typs mussten angepasst werden: Der Konstruktor für Ganzzahlkonstanten wurde von `Const` in `IntConst` umbenannt und es wurde ein Konstruktor `WordConst` für Wortkonstanten hinzugefügt. Zudem musste das letzte Argument der Konstruktoren `BinOp` (für binäre Operationen wie z. B. Addition), `UnOp` (für unäre Operationen wie z. B. Logarithmus) und `SetOp` (für Mengenoperationen wie z. B. Minimum) angepasst werden. Dieses ist jeweils die Funktion, die die Berechnung der Operation vornimmt. Der Typ dieses Arguments musste so geändert werden, dass der Rückgabetyp sowie die Parametertypen von `BinOp` und `UnOp` `term_types` anstatt von `int` sind. Hier kann man außerdem sehen, dass nun das Modul `ValueSet` für Mengen von Werten des Typs `term_types` anstatt von `IntSet` für Mengen von `int`-Werten verwendet wird. Für die Extraktion von Teilwörten wurde der Konstruktor `WordSlice` hinzugefügt, dessen Argumente der Term für das Wort und ein Tupel aus den Termen für die Start- und die Endposition des Teilwortes sind.

```
1 type intTerm = Const of int
2             | Param of string
3             | BinOp of string * intTerm * intTerm * (int -> int -> int)
4             | UnOp of string * intTerm * (int -> int)
5             | SetOp of string * symbSet * (IntSet.t -> int)
```

Listing 7: Der Typ `intTerm` vorher

```
1 type term = IntConst of int
2           | WordConst of string
3           | Param of string
4           | BinOp of string * term * term * (term_types -> term_types ->
    term_types)
5           | UnOp of string * term * (term_types -> term_types)
6           | SetOp of string * symbSet * (ValueSet.t -> term_types)
7           | WordSlice of term * (term * term)
```

Listing 8: Der Typ `term` jetzt

Analog dazu mussten auch die Typen `symbSet` für AST-Knoten von Mengenausdrücken, `bexpr` für Knoten von Booleanwerten in der Ausgabesprache und `propFormula` für instanziierte Formeln angepasst werden. Für Mengen wurde der Konstruktor `FiniteLanguage` für endliche Sprachen, die durch reguläre Ausdrücke beschrieben werden, hinzugefügt. In Listing 9 und Listing 10 ist die Definition des Typs vor bzw. nach der Änderung zu sehen.

```
1 type symbSet = SmallSet of intTerm list
2             | Enumeration of intTerm * intTerm * intTerm
3             | BinSetOp of string * symbSet * symbSet * (IntSet.t ->
    IntSet.t -> IntSet.t)
```

Listing 9: Der Typ `symbSet` vorher

```
1 type symbSet = SmallSet of term list
2             | Enumeration of term * term * term
3             | BinSetOp of string * symbSet * symbSet * (ValueSet.t ->
    ValueSet.t -> ValueSet.t)
4             | FiniteLanguage of regex
```

Listing 10: Der Typ `symbSet` jetzt

Änderung von Funktionen Außer Typen mussten auch Funktionen geändert werden, die noch den Typ `int` für Terme verwenden. Die Funktion `evalTerm` in der Datei `alschemes.ml` ist dafür zuständig, Terme auszuwerten, die als AST-Knoten vom Typ `term` vorliegen. Hierfür nutzt die Funktion Pattern-Matching, um festzustellen, welche Art von Term vorliegt.

```
1 let rec evalTerm eval alphabet = function
2  | IntConst(i)          -> Int i
3  | WordConst(w)         -> Word w
4  | Param(x)             -> (try ParamEval.find x eval
5                            with Not_found -> failwith ("Undefined parameter
     ‘" ^ x ^ "’"))
6  | BinOp(_,t,t’,f)      -> f (evalTerm eval alphabet t) (evalTerm eval
     alphabet q t’)
7  | UnOp(_,t,f)          -> f (evalTerm eval alphabet t)
8  | SetOp(_,m,f)         -> f (evalSet eval alphabet m)
9  | WordSlice(w,(s,e)) -> slice_of_word (evalTerm eval alphabet w) (
     evalTerm eval alphabet s) (evalTerm eval alphabet e)
```

Listing 11: Die Funktion `evalTerm` jetzt

Im Fall von Ganzzahlkonstanten (Konstruktor `IntConst`) musste der Wert mit dem Konstruktor `Int` von `term_types` gepackt werden. Für die Konstruktoren `BinOp`, `UnOp` und `SetOp` musste nichts geändert werden, da der Typ der Funktion im letzten Konstruktorparameter schon geändert wurde (s. Listing 8). Für Parameter (Konstruktor `Param`) wird der Wert des Parameters in der Parameter-Map `eval` gesucht. Da die Map bereits vorher gepackte Werte enthielt, musste hier das Entpacken des Wertes entfernt werden.

Die Funktion `evalSet` ist für die Auswertung von Iterationsmengen verallgemeinerter Junktoren vom Typ `symbSet` zuständig. Diese musste so geändert werden, dass die zurückgegebene Menge vom Typ `ValueSet.t` statt `IntSet.t` ist. Zudem mussten im Fall einer Aufzählung (Konstruktor `Enumeration`) die mittels `eval_term` ausgewerteten Terme entpackt werden. Analog zu `evalTerm` und `evalSet` mussten auch die entsprechenden Funktionen in der Implementierung der Ausgabesprache geändert werden.

Nach dem gleichen Schema musste auch die Funktion `instantiate`, die die Formeln instanziiert geändert werden: Werte von Termen mussten entpackt bzw. gepackt werden und für Mengen musste `ValueSet` anstatt von `IntSet` verwendet werden.

Die Funktionen `compareTerm` (vorher `compareIntTerm`) und `compareSymbSet`, deren Zweck es ist, AST-Knoten von Termen bzw. Mengen miteinander zu vergleichen, mussten so geändert werden, dass mit ihr Terme und Mengen, der neu hinzugefügten Konstruktoren verglichen werden können. Hierfür musste für jeden neuen Konstruktor in der Unterfunktion `order` eine natürliche Zahl als Ordnung festgelegt werden. Diese Ordnung wird verwendet, um Werte mit verschiedenen Konstruktoren miteinander zu vergleichen, wobei der Wert mit der größeren Ordnung als größer angenommen wird. In der Unterfunktion `comp` wird der Vergleich von zwei Werten derselben Ordnung und somit auch desselben Konstruktors implementiert.

Implementierung der Aufzählungen

Für die neu hinzugefügten Definitionsbereiche müssen nun die Aufzählungen (s. 3.1) implementiert werden. Die Aufzählungen in DiMo werden in der Datei `enumerators.ml` in einer Unterklasse der Klasse `enumerator` implementiert. Wird von `enumerator` geerbt,

muss ein Alphabet als Liste von OCaml-Strings an die Elternklasse übergeben werden. Diese Liste kann leer sein, wenn die Aufzählung kein Alphabet benötigt (z. B. wenn Ganzzahlen aufgezählt werden) oder wenn keine Zeichen abgeleitet werden können (z. B. bei dem regulären Ausdruck `SIGMA*`). Auf diese Liste kann mit der Methode `get_symbols` zugegriffen werden. Instanzen von `enumerator` werden im Folgenden als *Enumerator* bezeichnet. Folgende Methoden von `enumerator` sind als `virtual` deklariert und müssen somit überschrieben werden:

- `get_name` gibt den Namen des Parameters zurück, dessen Definitionsbereich aufgezählt wird. Dieser ist in der Regel ein Klassenparameter mit dem Namen `x`.
- `is_infinite` gibt `true` zurück, falls die aufgezählte Menge endlich ist, ansonsten `false`.
- `how_many` gibt die Kardinalität der aufzuzählenden Menge zurück.
- `get` implementiert die eigentliche Aufzählung nach Definition 18. Somit wird dieser Methode eine Ganzzahl als Parameter übergeben. Der Rückgabewert ist vom Typ `enum_result option`, wobei ein Wert von `None` für `None` nach Definition 18 steht und `Some v` für den Wert `v` aus der aufzuzählenden Menge.

Folgende Methoden sind zwar in der Elternklasse deklariert, müssen aber bei Bedarf überschrieben werden:

- `needs_alphabet` Diese Methode gibt einen Boolean zurück, der angibt, ob der Enumerator ein explizit angegebenes Alphabet benötigt.
- `set_alphabet` Diese Methode wird aufgerufen, wenn das (explizite oder implizite) Alphabet feststeht. Sie nimmt als Parameter das Alphabet als `ValueSet` entgegen und überprüft mittels `needs_alphabet`, ob das übergebene Alphabet leer ist und der Enumerator ein explizit angegebenes Alphabet benötigt. In dem Fall wird das Programm nach 3.1 mit einer Fehlermeldung beendet.

Für endliche Wortmengen, bei denen die Elemente explizit im Header angegeben wurden, wurde die Klasse `finiteWordSetEnumerator` erstellt, die analog zu `finiteSetEnumerator` für explizit angegebene Zahlenmengen alle Elemente in einem Array speichert und nacheinander aufzählt. Für Definitionsbereiche, die durch reguläre Ausdrücke beschrieben sind, wurde für die naive Aufzählung die Enumerator-Klasse `naiveRegexEnumerator` erstellt. Für die zusammengesetzte Aufzählung wurde für jede Aufzählungsart eine eigene Enumerator-Klasse erstellt:

- `composedAlphabetRegexEnumerator` für den regulären Ausdruck `SIGMA`
- `composedUnionRegexEnumerator` für Vereinigungen
- `composedConcatRegexEnumerator` für Konkatenationen
- `composedStarRegexEnumerator` für die kleenesche Hülle
- `composedPlusRegexEnumerator` für die positive Hülle

Für die direkte Aufzählung wurde die Enumerator Klasse `directRegexEnumerator` erstellt. Die Implementierung der Length-Ordered-Lists ist in die Datei `lazySet.ml` ausgelagert. Der Namensbestandteil „`lazy`“ des Dateinamens kommt daher, dass für die Implementierung der Length-Ordered-Lists Lazy Evaluation genutzt werden musste, um unendliche Mengen darstellen zu können [8].

Implementierung des Alphabets

In der Funktion `makeEnumerator` in der Datei `alschemes.ml` wird der Enumerator erstellt. Ein Ausschnitt dieser Funktion ist in Listing 12 zu sehen. Dort wird in Zeile 1 die Unterfunktion `mkEnum` aufgerufen, die über alle im Header angegebenen Parameter iteriert und daraus einen Enumerator konstruiert, der das Kreuzprodukt der Definitionsbereiche aller Parameter aufzählt. Am Ende wird in Zeile 3 auf dem konstruierten Enumerator die Methode `set_alphabet` aufgerufen. Das Argument dieser Methode hängt davon ab, ob explizit ein Alphabet angegeben wurde. Dies ist der Fall, wenn beim Durchsuchen der Parameter einer mit dem Namen `SIGMA` und einem Definitionsbereich vom Konstruktor `AlphabetSet` gefunden wurde. Wurde kein solcher Parameter gefunden (Zeile 5), wird das Alphabet nun erstellt, indem in der von `enum#get_symbols` zurückgegebenen Liste jedes Symbol gepackt wird und die Liste anschließend in eine Menge umgewandelt wird. Falls das Alphabet angegeben wurde, wird in Zeile 6 mit `check_alphabets` sichergestellt, dass alle Symbole des Enumerators auch im Alphabet auftauchen. Ansonsten wird das Programm mit einer Fehlermeldung beendet.

```
1 let enum = mkEnum params
2 in
3 enum#set_alphabet (
4   match !explicit_alphabet with
5     | None -> ValueSet.of_list (List.map pack_word enum#get_symbols)
6     | Some alphabet -> check_alphabets enum#get_symbols alphabet
7 );
8 enum
```

Listing 12: Hier wird das implizite/explizite Alphabet festgelegt

4 Anwendungsfälle

In diesem Kapitel werden Beispielprobleme gezeigt, die veranschaulichen, wie die neu ergänzten Features von DiMo genutzt werden können. Alle diese Beispielprobleme sind also so modelliert, dass sie mindestens einen Parameter mit Worttyp haben.

4.1 Teilwort-Problem

Das erste Problem, welches modelliert wurde, ist Folgendes:

Problem 1

gegeben: Wort $u \in L_u$ und Wort $v \in L_v$

Frage: Ist u Teilwort von v, also gibt es ein $j \in \{0, \dots, |v| - |u|\}$, sodass für alle $i \in \{1, \dots, |u|\}$ gilt: $u(i) = v(j - 1 + i)$?

Dieses Problem wurde folgendermaßen in DiMo beschrieben:

```
1 SATISFIABLE IsSubword(u, v)
2 PROPOSITIONS S
3 PARAMETERS u: {"ab", "abc"},
4            v: COMPOSED "["_("" | (("’"_("a" | "b"_("a" | "b")*)_"’")_(",
    ’"_("a" | "b"_("a" | "b")*)_"’")*))_"]",
5            SIGMA: {" ", "’", ",", "a", "b", "[", "]"}
```

In Zeile 3 wird $L_u = \{a, abc\}$ festgelegt. In Zeile 4 wird L_v mit einem regulären Ausdruck beschrieben. Dieser reguläre Ausdruck beschreibt die Sprache aller Wörter, die eine Liste aller Wörter aus den Zeichen a und b, die entweder aus nur einem a bestehen oder mit einem b beginnen, in Python-Syntax (mit einfachen Anführungszeichen und einem

Leerzeichen nach jedem Komma) darstellen, z. B. ist $v = ['a','ba'] \in L_v$. Ist diese Formel erfüllbar, so bedeutet dies, dass das Wort u (möglicherweise als Teilwort) in der Liste, die durch v beschrieben ist, enthalten ist.
Das Variablensymbol S wird genutzt, um die Startposition zu codieren, ab der u in v zu finden ist. $S(i)$ soll also genau dann 1 sein, wenn u ab der Position i in v startet. Die Definitionen der Hilfsformeln sehen folgendermaßen aus:

```
1  IsSubword(u, v) = OnlyOneStartPosition(v) & MapPropositionsU(u) &
   MapPropositionsV(v) & SubwordCondition(u, v)
2
3  OnlyOneStartPosition(v) = FORALL i: {1,..,|v|}. S(i) -> FORALL i':
   {1,..,|v|}-{i}. -S(i')
4
5  MapPropositionsU(u) = FORALL i: {1,..,|u|}. U(i, u(i)) & FORALL s:
   SIGMA-{u(i)}. -U(i, s)
6
7  MapPropositionsV(v) = FORALL i: {1,..,|v|}. V(i, v(i)) & FORALL s:
   SIGMA-{v(i)}. -V(i, s)
8
9  SubwordCondition(u, v) = FORALL s: SIGMA. U(1, s) -> FORSOME j:
   {1,..,|v|-|u|+1}. S(j) & V(j, s)
10                                              & FORALL i: {2,..,|u|}.
   FORALL s': SIGMA. U(i, s')
11                                              -> V(i+j-1, s')
```

Mit der Formel `OnlyOneStartPosition` wird sichergestellt, dass es nur eine Startposition gibt. Das bedeutet nicht, dass u nicht mehrfach als Teilwort in v auftreten darf, sondern dass bei mehrfachem Auftreten nur eine Position als Ergebnis ausgegeben wird. Durch die Formeln `MapPropositionsU` und `MapPropositionsV` wird festgelegt, dass die Hilfsvariablen $U(i,s)$ bzw. $V(i,s)$ für $s \in \Sigma$ genau dann mit 1 belegt werden sollen, wenn s das i-te Zeichen in u bzw. v ist. `SubwordCondition` ist schließlich die Formel, die die eigentliche Teilwort-Eigenschaft modelliert. In Listing 13 ist der Anfang der Ausgabe von DiMo bei Ausführung auf diesem Beispielproblem zu sehen.

```
done.
Opening input file `./examples/version 0.3/subwords.dm´  done.
Constructing engine for formula scheme .....
  IsSubword(u,v)
Running engine ...............................

Running iteration 0
  Constructing parameter evaluation ........... u="ab", v="[]"
  Alphabet .................................... SIGMA={" ", "'", ",", "[", "]", "a", "b", "c"}
  Getting new solver .......................... done.
  Adding clauses .............................. done.
  Solving ..................................... unsatisfiable!
Instance u="ab", v="[]" ....................... unsatisfiable.

Running iteration 1
  Constructing parameter evaluation ........... u="abc", v="[]"
  Alphabet .................................... SIGMA={" ", "'", ",", "[", "]", "a", "b", "c"}
  Getting new solver .......................... done.
  Adding clauses .............................. done.
  Solving ..................................... unsatisfiable!
Instance u="abc", v="[]" ...................... unsatisfiable.

Running iteration 2
  Constructing parameter evaluation ........... u="ab", v="['a']"
  Alphabet .................................... SIGMA={" ", "'", ",", "[", "]", "a", "b", "c"}
  Getting new solver .......................... done.
  Adding clauses .............................. done.
  Solving ..................................... unsatisfiable!
Instance u="ab", v="['a']" .................... unsatisfiable.
```

Listing 13: Ausgabe der Ausführung von DiMo auf dem Teilwort-Problem

4.2 Teilfolgen-Problem

Problem 2

gegeben: Wort $u \in L_u$ und Wort $v \in L_v$

Frage: Ist u eine Teilfolge von v, also gibt es eine Folge von Indizes $s_1, \ldots, s_{|u|}$ mit $s_i < s_{i+1}$, sodass $u(i) = v(s_i)$ für alle $i \in \{1, \ldots, |u|\}$?

Dieses Problem wurde so modelliert:

```
1 SATISFIABLE IsSubsequence(u, v)
2 PROPOSITIONS S
3 PARAMETERS u: COMPOSED "’"_("a" | "b"_("a" | "b")*)_"’",
4            v: DIRECT "["_("" | (("’"_("a" | "b"_("a" | "b")*)_"’")_(", ’"
   _("a" | "b"_("a" | "b")*)_"’")*))_"]"
5 FORMULAS
```

Die Definitionsbereiche L_u und L_v wurden diesmal beide mit regulären Ausdrücken beschrieben. L_u ist die Menge aller Wörter aus den Zeichen a und b, die nur aus einem a bestehen oder mit einem b beginnen. L_v ist so definiert wie beim Teilwort-Problem. Diesmal ist das Alphabet nicht explizit angegeben. Mit dem Variablensymbol S wird die oben beschriebene Folge (s_i) modelliert. $S(i, j)$ soll sich also genau dann zu 1 auswerten, wenn $s_i = j$ gilt. Die Hilfsformeln, die zur Modellierung dieses Problems verwendet wurden, sind folgendermaßen definiert:

```
1
2     ForEveryLeftAtLeastOneRight(u, v) = FORALL i: {1,..,|u|}. FORSOME j:
    {i,..,|v|}. S(i, j) & IsEqual(u(i), v(j))
3
4     ForEveryLeftAtMostOneRight(u, v) =
5         FORALL i: {1,..,|u|}.
6         FORALL j: {i,..,|v|}. S(i, j) -> FORALL j’: {1,..,|v|}-{j}. -S(i,
     j’)
7
8     StrictlyMonotone(u, v) =
9         FORALL i: {1,..,|u|}.
10        FORALL j: {i,..,|v|}.
11            S(i, j) -> FORALL i’: {1,..,i-1}. FORSOME j’: {1,..,j-1}. S(i
    ’, j’)
12
13    IsEqual(a, b) = FORALL e: {a}-{b}. False
```

`ForEveryLeftAtLeastOneRight` und `ForEveryLeftAtMostOneRight` stellen sicher, dass es für jedes $i \in \{1, \ldots, |u|\}$ genau ein $j \in \{1, \ldots, |v|\}$ gibt, sodass $\mathcal{I}(S(i, j)) = 1$ gilt (also dass die Folge (s_i) eine totale und nicht-mehrwertige Funktion ist). Mit `StrictlyMonotone` wird umgesetzt, dass die Folge (s_i) streng monoton steigend ist. `IsEqual` ist eine Hilfsformel, die sich immer genau dann zu 1 auswertet, wenn ihre beiden Parameter a und b gleich sind. Hierbei wird der Fakt ausgenutzt, dass eine Konjunktion über die leere Menge immer wahr ist. Die Menge, über die der verallgemeinerte Junktor geht, ist genau dann leer, wenn $a = b$ gilt. Ist dies nicht der Fall, ist die Menge nicht leer und die Formel wertet sich offensichtlich zu 0 aus.

```
done.
Opening input file ‘./examples/version 0.3/subsequence.dm´  done.
Constructing engine for formula scheme .....
  IsSubsequence(u,v)
Running engine ...............................

Running iteration 0
  Constructing parameter evaluation ........... u="’a’", v="[]"
  Alphabet .................................... SIGMA={" ", "’", ",", "[", "]", "a", "b"}
  Getting new solver .......................... done.
  Adding clauses .............................. done.
  Solving ..................................... unsatisfiable!
Instance u="’a’", v="[]" ...................... unsatisfiable.
```

```
Running iteration 1
  Constructing parameter evaluation ........... u="'b'", v="[]"
  Alphabet ................................... SIGMA={" ", "'", ",", "[", "]", "a", "b"}
  Getting new solver .......................... done.
  Adding clauses .............................. done.
  Solving ..................................... unsatisfiable!
Instance u="'b'", v="[]" ...................... unsatisfiable.

Running iteration 2
  Constructing parameter evaluation ........... u="'a'", v="['a']"
  Alphabet ................................... SIGMA={" ", "'", ",", "[", "]", "a", "b"}
  Getting new solver .......................... done.
  Adding clauses .............................. done.
  Solving ..................................... satisfiable!
  Collecting relevant literals from solution ... done.
Instance u="'a'", v="['a']" ................... satisfiable.
  Satisfying assignment: -S(1,1), S(1,2), -S(1,3), -S(1,4), -S(1,5), -S(2,1), -S(2,2), S(2,3), -S(2,4), -S
     (2,5), -S(3,1), -S(3,2), -S(3,3), S(3,4), -S(3,5)
```

Listing 14: Ausgabe der Ausführung von DiMo auf dem Teilfolgen-Problem

4.3 Präfix-Problem

Problem 3

gegeben: Wort $u \in L_u$ und Wort $v \in L_v$
Frage: Ist u Präfix von v, also gilt

$$|u| \leq |v| \tag{1}$$
$$u(i) = v(i) \text{ für alle } i \in \{1, \dots, |u|\}? \tag{2}$$

Dieses Problem wurde so modelliert:

```
1  SATISFIABLE IsPrefix(u, v)
2  PROPOSITIONS U, V
3  PARAMETERS u: COMPOSED ("" | "a")^2, v: COMPOSED "a"_("a" | "b" | "c")
4  FORMULAS
5      IsPrefix(u, v) = MapPropositionsU(u) & MapPropositionsV(v) &
     LessEqual(|u|, |v|) & PrefixCondition(u, v)
6
7      MapPropositionsU(u) = FORALL i: {1,..,|u|}. U(i, u(i)) & FORALL s:
     SIGMA-{u(i)}. -U(i, s)
8
9      MapPropositionsV(v) = FORALL i: {1,..,|v|}. V(i, v(i)) & FORALL s:
     SIGMA-{v(i)}. -V(i, s)
10
11     LessEqual(a, b) = FORSOME s: {0,..,MAX{-1, b-a}}. True
12
13     PrefixCondition(u, v) = FORALL i: {1,..,|u|}. V(i, u(i))
```

`MapPropositionsU` und `MapPropositionsV` sind hier genauso definiert, wie bei dem Teilwort-Problem. Mit der Formel `LessEqual` wird (1) sichergestellt. `PrefixCondition` ist genau dann wahr, wenn (2) zutrifft.

```
done.
Opening input file `./examples/version 0.3/prefix.dm´ . done.
Constructing engine for formula scheme .....
  IsPrefix(u,v)
Running engine ...............................

Running iteration 0
  Constructing parameter evaluation ........... u="", v="aa"
  Alphabet ................................... SIGMA={"a", "b", "c"}
  Getting new solver .......................... done.
  Adding clauses .............................. done.
  Solving ..................................... satisfiable!
  Collecting relevant literals from solution ... done.
Instance u="", v="aa" ......................... satisfiable.
  Satisfying assignment: V(1,"a"), -V(1,"b"), -V(1,"c"), V(2,"a"), -V(2,"b"), -V(2,"c")

Running iteration 1
  Constructing parameter evaluation ........... u="", v="ab"
  Alphabet ................................... SIGMA={"a", "b", "c"}
```

```
  Getting new solver .......................... done.
  Adding clauses .............................. done.
  Solving ..................................... satisfiable!
  Collecting relevant literals from solution ... done.
Instance u="", v="ab" ......................... satisfiable.
  Satisfying assignment: V(1,"a"), -V(1,"b"), -V(1,"c"), -V(2,"a"), V(2,"b"), -V(2,"c")

Running iteration 2
  Constructing parameter evaluation ........... u="", v="ac"
  Alphabet .................................... SIGMA={"a", "b", "c"}
  Getting new solver .......................... done.
  Adding clauses .............................. done.
```

Listing 15: Ausgabe der Ausführung von DiMo auf dem Präfix-Problem

4.4 Levenshtein-Distanz-Problem

Problem 4

gegeben: Wort w und Zahl n

Frage: Ist n die Levenshtein-Distanz von w zu einer festgelegten Sprache L?

Um dieses Problem zu lösen, muss zuallererst die Levenshtein-Distanz zwischen 2 Wörtern u, v definiert werden. Diese beschreibt die minimale Anzahl an Operationen, um u in v zu umzuwandeln. Eine Operation kann die Entfernung, Einfügung oder Ersetzung eines Zeichens sein [12].

Definition 23. Seien $u, v \in \Sigma^*$ Wörter über Σ. Dann ist die **Levenshtein-Distanz** $\text{lev}(u, v)$ zwischen u und v folgendermaßen definiert [13]:

$$\text{lev}(u,v) = \begin{cases} |u|, & \text{falls } v = \varepsilon \\ |v|, & \text{falls } u = \varepsilon \\ \text{lev}(u(2,|u|), v(2,|v|)), & \text{falls } u(1) = v(1) \\ 1 + \min \left\{ \begin{array}{ll} \text{lev}(u(2,|u|), v) & (u(1) \text{ wird entfernt}) \\ \text{lev}(u, v(2,|v|)) & (v(1) \text{ wird hinzugefügt}) \\ \text{lev}(u(2,|u|), v(2,|v|)) & (u(1) \text{ wird durch } v(1) \text{ ersetzt}) \end{array} \right\} & \text{sonst} \end{cases}$$

Nun folgt noch ein Beispiel:

Beispiel 5. Für die Levenshtein-Distanz der Wörter $u = ab$ und $v = bca$ gilt:

$$\begin{aligned} \text{lev}(u,v) = \text{lev}(ab, bca) &= 1 + \min \left\{ \begin{array}{ll} \text{lev}(u(2,|u|), v) & (u(1) \text{ wird entfernt}) \\ \text{lev}(u, v(2,|v|)) & (v(1) \text{ wird hinzugefügt}) \\ \text{lev}(u(2,|u|), v(2,|v|)) & (u(1) \text{ wird durch } v(1) \text{ ersetzt}) \end{array} \right\} \\ &= 1 + \min \left\{ \begin{array}{ll} \text{lev}(b, bca) & (a \text{ wird entfernt}) \\ \text{lev}(ab, ca) & (b \text{ wird hinzugefügt}) \\ \text{lev}(b, ca) & (a \text{ wird durch } b \text{ ersetzt}) \end{array} \right\} \end{aligned}$$

Nun müssen $\text{lev}(b, bca)$, $\text{lev}(ab, ca)$ und $\text{lev}(b, ca)$ rekursiv berechnet werden:

$$\begin{aligned} \text{lev}(b, bca) &= \text{lev}(\varepsilon, ca) \quad \text{da jeweils das erste Zeichen von } b \text{ und } bca \text{ gleich ist} \\ &= |ca| \\ &= 2 \end{aligned}$$

$$\text{lev}(ab, ca) = 1 + \min \left\{ \begin{array}{ll} \text{lev}(b, ca) & (a \text{ wird entfernt}) \\ \text{lev}(ab, a) & (c \text{ wird hinzugefügt}) \\ \text{lev}(b, a) & (a \text{ wird durch } c \text{ ersetzt}) \end{array} \right\}$$

Für die Berechnung von $\text{lev}(ab, a)$ gilt:

$$\begin{aligned}\text{lev}(ab, a) &= \text{lev}(b, \varepsilon)\\ &= |b|\\ &= 1\end{aligned}$$

Die Berechnungen von $\text{lev}(b, ca) = 2$ und $\text{lev}(b, a) = 1$ erfolgen analog und werden der Übersichtlichkeit halber weggelassen. Somit ergibt sich 1 als Minimum von $\text{lev}(b, ca)$, $\text{lev}(ab, a)$ und $\text{lev}(b, a)$, sodass:

$$\text{lev}(ab, ca) = 1 + 1 = 2$$

Wie erwähnt gilt $\text{lev}(b, ca) = 2$, sodass sich nach Einsetzen der Werte Folgendes für $\text{lev}(u, v)$ ergibt:

$$\begin{aligned}\text{lev}(u, v) &= 1 + \min \left\{ \begin{array}{ll} \text{lev}(b, bca) & (a \text{ wird entfernt}) \\ \text{lev}(ab, ca) & (b \text{ wird hinzugefügt}) \\ \text{lev}(b, ca) & (a \text{ wird durch } b \text{ ersetzt}) \end{array} \right\}\\ &= 1 + \min\{2, 2, 2\}\\ &= 1 + 2\\ &= 3\end{aligned}$$

Dieses Ergebnis bedeutet also, dass mindestens drei Operationen nötig sind, um $u = ab$ in $v = bca$ umzuwandeln. Eine Umwandlungsmöglichkeit, die die minimale Anzahl an Operationen braucht, lässt sich daraus ableiten, welcher der Werte im letzten Fall von Definition 23 minimal war. Wenn mehrere Werte davon minimal sind (oben hatten beispielsweise alle drei Werte den Wert 2), gibt es mehrere Möglichkeiten, um u mit der minimalen Anzahl an Operationen in v umzuwandeln. In diesem Fall ist eine Möglichkeit zur Umwandlung mit drei Operationen:

$$ab \xrightarrow{\text{entferne } a} b \xrightarrow{\text{füge } c \text{ hinzu}} bc \xrightarrow{\text{füge } a \text{ hinzu}} bca$$

Die Levenshtein-Distanz eines Wortes u zu einer Sprache L ist dann die minimale Levenshtein-Distanz, die w zu einem Wort aus L haben kann:

$$\text{lev}(w, L) = \min\{\text{lev}(w, w') \mid w' \in L\}$$

Dieses Problem wurde folgendermaßen in DiMo modelliert:

```
1  SATISFIABLE HasDistanceToLanguange(w, n)
2  PROPOSITIONS W
3  PARAMETERS w: {"a", "ab"}, n: {0, 1}
4  FORMULAS
5      HasDistanceToLanguange(w, n) = WordWithDistanceExists(w, n) &
       NoWordWithLowerDistance(w, n)
6
7      WordWithDistanceExists(w, n) =
8          FORSOME l: {MAX {0, |w|-n},..,|w|+n}.
9          FORSOME w': REG ((SIGMA-"a") | "a"_(SIGMA-"a")*_"a")* & SIGMA^l.
       (* Replace with arbitrary regex *)
10             W(w')
11             & HasDistanceToWord(w, w', n)
12
13     NoWordWithLowerDistance(w, n) =
```

```
14          FORALL n': {0,..,n-1}.
15          FORALL l: {MAX {0, |w|-n'},..,|w|+n'}.
16          FORALL w': REG ((SIGMA-"a") | "a"_(SIGMA-"a")*_"a")* & SIGMA^l.
    (* Replace with arbitrary regex *)
17              -HasDistanceToWord(w, w', n')
18
19     HasDistanceToWord(u, "", n) = IsEqual(|u|, n)
20
21     HasDistanceToWord("", v, n) = IsEqual(|v|, n)
22
23     HasDistanceToWord(u, v, n) =
24           (IsEqual(u(1), v(1)) & HasDistanceToWord(u(2,..), v(2,..), n))
25         | (-IsEqual(u(1), v(1)) & ((HasDistanceToWord(u(2,..), v, n-1)
26                                   & FORALL n': {0,..,n-2}. -
    HasDistanceToWord(u, v(2,..), n')
27                                                          & -
    HasDistanceToWord(u(2,..), v(2,..), n'))
28                                  | (HasDistanceToWord(u, v(2,..), n-1)
29                                   & FORALL n': {0,..,n-2}. -
    HasDistanceToWord(u(2,..), v, n')
30                                                          & -
    HasDistanceToWord(u(2,..), v(2,..), n'))
31                                  | (HasDistanceToWord(u(2,..), v(2,..), n
    -1)
32                                   & FORALL n': {0,..,n-2}. -
    HasDistanceToWord(u(2,..), v, n')
33                                                          & -
    HasDistanceToWord(u, v(2,..), n'))))
```

In Zeile 2 wird W als Variablensymbol festgelegt. Zweck dieses Symbols ist, dass in einem Modell für mindestens ein $w' \in L$ mit $\text{lev}(w, w') = n$ die Variable $W_{w'}$ auf 1 gesetzt ist. So kann, falls die Formel erfüllbar ist, direkt aus dem Modell mindestens ein Wort aus L abgeleitet werden, welches die minimale Distanz zu w hat. Die Sprache L wird hier über den regulären Ausdruck in den Zeilen 9 und 16 festgelegt. In diesem Beispiel ist L die Menge aller Wörter, die eine gerade Anzahl von a enthalten. Dieser reguläre Ausdruck wird im Folgenden als α bezeichnet. Der Ansatz hierbei war, dass w genau dann die Distanz n zu L hat, wenn folgende Bedingungen zutreffen:

(1) Es gibt ein $w' \in L$ mit $\text{lev}(w, w') = n$.

(2) Für alle $n' < n$ und alle $w' \in L$ gilt $\text{lev}(w, w') \neq n'$.

(1) wird mit `WordWithDistanceExists` sichergestellt. Formal ausgedrückt ist die Formel:

$$\varphi_{\texttt{WordWithDistanceExists}}(w, n) = \bigvee_{l=\max\{0,|w|-n\}}^{|w|+n} \bigvee_{w' \in L \cap \Sigma^l} W_{w'} \wedge \varphi_{\texttt{HasDistanceToWord}}(w, w', n)$$

Hierbei wird für alle Wörter aus L, deren Länge maximal um n von $|w|$ abweicht, überprüft, ob diese die Distanz n zu w haben. Alle Wörter, deren Länge mehr als n von $|w|$ abweicht, haben definitiv eine Distanz, die größer als n ist (da auf jeden Fall mehr als n Zeichen hinzugefügt werden müssen). Somit müssen diese Wörter nicht mehr überprüft werden. Durch die Konjunktion mit der Variable $W_{w'}$, hat $W_{w'}$ immer den Wert 1, wenn w' die Distanz n zu w hat. Es ist auch möglich, dass es ein Modell der Formel gibt, bei dem $W_{w'}$ den Wert 1 hat, obwohl w' nicht eine Distanz von n zu w hat. Allerdings filtert DiMo solche Variablensymbole heraus, da sich $\varphi_{\texttt{HasDistanceToWord}}(w, w', n)$ und somit auch die Konjunktion $W_{w'} \wedge \varphi_{\texttt{HasDistanceToWord}}(w, w', n)$ zu 0 auswerten, sodass am Ende nur

Variablensymbole für Wörter ausgegeben werden, die eine Distanz von n zu w haben. Die Formel `NoWordWithLowerDistance` stellt (2) sicher, indem für alle $n' < n$ mit demselben Verfahren überprüft wird, dass es kein Wort w' gibt, welches eine Distanz von n' zu w hat:

$$\varphi_{\texttt{NoWordWithLowerDistance}}(w,n) = \bigwedge_{n'=0}^{n-1} \bigwedge_{l=\max\{0,|w|-n'\}}^{|w|+n'} \bigwedge_{w'\in L\cap\Sigma^l} \neg\varphi_{\texttt{HasDistanceToWord}}(w,w',n')$$

`HasDistanceToWord` überprüft schließlich für zwei Wörter u und v, ob diese die Distanz n haben. Diese Formel ist rekursiv nach Definition 23 aufgebaut. Die Basisfälle, bei denen eines der Wörter leer ist, sind in Zeile 19 und 21 implementiert. In diesen Fällen wertet sich die Formel genau dann zu 1 aus, wenn n die Länge des anderen Wortes ist. Die rekursiven Fälle sind ab Zeile 23 implementiert. In Zeile 24 wird der Fall behandelt, dass das jeweils erste Zeichen von u und v gleich ist. In dem Fall muss die Distanz zwischen $u(2,|u|)$ und $v(2,|v|)$ genau n sein. Bei allen anderen Fällen muss das Ausgangswort verändert werden. In Zeile 25 ist der Fall, dass $u(1)$ entfernt wird, abgedeckt. Die Distanz zwischen $u(2,|u|)$ und v muss hier $n-1$ betragen, da in Definition 23 auf die Distanz 1 addiert wird. Zeile 26 und 27 sorgen dafür, dass die Distanz, die durch Entfernung von $u(1)$ entsteht, auch die minimale Distanz ist. Es darf also kein $n' < n-1$ geben, sodass n' die Distanz ist, die durch Einfügen oder Ersetzen des ersten Zeichens entsteht. Analog dazu werden auch die Fälle behandelt, wo das erste Zeichen eingefügt (ab Zeile 28) oder ersetzt (ab Zeile 31) wird. In Listing 16 ist die Ausgabe von DiMo bei Ausführung auf diesem Problem zu sehen. Darin lässt sich ablesen, dass für $w = a$ und $n = 1$ die Formel erfüllbar ist. An der Ausgabe des Modells in Zeile 29 ist erkennbar, dass W_{aa} auf 1 gesetzt ist, woraus sich ableiten lässt, dass $w' = abc$ ein Wort aus L mit $\text{lev}(w,w') = n$ ist.

```
done.
Opening input file `./examples/version 0.3/levenshtein.dm´  done.
Constructing engine for formula scheme .....
  HasDistanceToLanguange(w,n)
Running engine ................................

Running iteration 0
  Constructing parameter evaluation ........... n=0, w="a"
  Alphabet .................................... SIGMA={"a", "b"}
  Tidying yields .............................. False
  Collecting propositions .....................
  Transformation to CNF yields ................ {{}}
  Getting new solver .......................... done.
  Adding clauses .............................. done.
  Solving ..................................... unsatisfiable!
Instance n=0, w="a" ........................... unsatisfiable.

Running iteration 1
  Constructing parameter evaluation ........... n=1, w="a"
  Alphabet .................................... SIGMA={"a", "b"}
  Tidying yields .............................. W("aa") | W("b") | W("")
  Collecting propositions ..................... W(""), W("aa"), W("b")
  Transformation to CNF yields ................ { { W("aa"), W("b"), W("") } }
  Getting new solver .......................... done.
  Adding clauses .............................. done.
  Solving ..................................... satisfiable!
  Collecting relevant literals from solution ... done.
Instance n=1, w="a" ........................... satisfiable.
  Satisfying assignment: -W(""), W("aa"), -W("b")

Running iteration 2
  Constructing parameter evaluation ........... n=0, w="ab"
  Alphabet .................................... SIGMA={"a", "b"}
  Tidying yields .............................. False
  Collecting propositions .....................
  Transformation to CNF yields ................ {{}}
  Getting new solver .......................... done.
  Adding clauses .............................. done.
  Solving ..................................... unsatisfiable!
Instance n=0, w="ab" .......................... unsatisfiable.

Running iteration 3
  Constructing parameter evaluation ........... n=1, w="ab"
  Alphabet .................................... SIGMA={"a", "b"}
  Tidying yields .............................. W("aab") | W("aba") | W("aa") | W("bb") | W("b")
  Collecting propositions ..................... W("aa"), W("aab"), W("aba"), W("b"), W("bb")
  Transformation to CNF yields ................ { { W("aab"), W("aba"), W("aa"), W("bb"), W("b") } }
```

```
  Getting new solver ........................... done.
  Adding clauses ............................... done.
  Solving ...................................... satisfiable!
  Collecting relevant literals from solution ... done.
Instance n=1, w="ab" ........................... satisfiable.
  Satisfying assignment: -W("aa"), W("aab"), -W("aba"), -W("b"), -W("bb")

Running iteration 4
  Constructing parameter evaluation ...........
  Alphabet ..................................... SIGMA={"a", "b"}
stopped.
```

Listing 16: Ausgabe der Ausführung von DiMo auf dem Levenshtein-Distanz-Problem

5 Fazit

Abschließend wird nun darauf eingegangen, welche Probleme nach der Implementierung offen geblieben sind und welche weiteren Erweiterungen an DiMo noch vorgenommen werden können.

5.1 Offene Probleme

Bei der in dieser Arbeit umgesetzten Implementierung bestehen noch einige Probleme: Bisher ist es noch nötig, Wortkonstanten in regulären Ausdrücken in Anführungszeichen zu schreiben, wodurch die Ausdrücke unter Umständen sehr lang und unübersichtlich werden. Mit Anpassungen im Lexer und Parser könnte dies geändert werden, sodass nur noch spezielle Zeichen wie Leerzeichen, Anführungszeichen oder Klammern in Anführungszeichen stehen müssten. So könnte der Ausdruck aus dem Teilwort-Problem (s. Listing 4) von

```
1    "["_("" | (("'"_("a" | "b"_("a" | "b")*)_"'")_(", '"_("a" | "b"_("a"
     | "b")*)_"'")*))_"]"
```

zu

```
1    [_("" | (("'"_(a | b_(a | b)*)_"'")_(", '"_(a | b_(a | b)*)_"'")*))_]
```

verkürzt werden.
Eine implementierungsbedingte Einschränkung entsteht dadurch, dass jedes Zeichen in einem Wort durch ein Byte repräsentiert ist. Das führt dazu, dass das Alphabet auf maximal 256 Zeichen beschränkt ist, da ein Byte, welches aus 8 Bit zusammengesetzt ist, nur $2^8 = 256$ unterschiedliche Werte annehmen kann. Ist die Eingabedatei zudem in UTF-8 codiert, werden Zeichen, die nicht im ASCII-Zeichensatz enthalten sind (z. B. Umlaute oder das Paragrafzeichen), durch mehrere Bytes repräsentiert [14], sodass diese von DiMo als mehrere Zeichen erkannt werden. Dadurch kann es zu schwer verständlichen Fehlermeldungen kommen, wenn ein solches Zeichen explizit im Alphabet angegeben wird.
Ein weiteres Problem ist noch, dass bisher nur für die zusammengesetzte Aufzählung eine Abschätzung der Kardinalität der Ergebnissprache implementiert ist (s. Definition 19). Für die naive und direkte Aufzählung werden stattdessen nur unendliche Sprachen als unendlich erkannt. Der Wert für die Kardinalität wird nicht nur innerhalb der Implementierung der zusammengesetzten Aufzählung verwendet, sondern auch bei der Aufzählung von dem Kreuzprodukt der Definitionsbereiche, wenn mehrere Parameter im Header angegeben sind. Dies führt dazu, dass das Programm bei mehreren Parametern mit einer Fehlermeldung beendet werden kann, wenn ein Wortparameter einen endlichen Definitionsbereich hat, der naiv oder direkt aufgezählt wird. Zu einem solchen Fehler kommt es immer dann, wenn dieser Parameter nicht der erste im Header angegebene Parameter ist.

Auch die Erkennung, ob eine reguläre Sprache endlich oder unendlich ist, ist bisher nur eine Schätzung, wobei eine endliche Sprache fälschlicherweise als unendlich erkannt werden kann (aber nie eine unendliche Sprache als endlich). Zu Fehleinschätzungen kommt es für Ausdrücke der Form α^* oder α^+, sofern $L(\alpha) = \{\varepsilon\}$ gilt, oder für Ausdrücke $\alpha_1 \cdot \alpha_2$, falls einer der Unterausdrücke α_1 die leere Sprache beschreibt und der andere eine unendliche. Dieses Problem kann gelöst werden, da für eine reguläre Sprache entscheidbar ist, ob diese unendlich ist, indem beispielsweise aus dem Ausdruck ein endlicher Automat generiert wird [15, S. 293].

5.2 Mögliche Erweiterungen

Eine Erweiterungsmöglichkeit wäre, eine bessere Aufzählung für reguläre Sprachen hinzuzufügen. Z. B. könnte man die Aufzählung aus [8] implementieren, die nichtdeterministische endliche Automaten nutzt.
Außerdem lassen sich die implementierten Aufzählungen noch optimieren, indem bereits aufgezählte Wörter zwischengespeichert werden (z. B. in Hashtabellen). Dies würde insbesondere zu Zeitersparnissen führen, wenn ein Wort bei der Aufzählung mehrerer Parameter mehrmals aufgezählt wird.
Eine weitere Möglichkeit, DiMo zu erweitern, wäre einen Typechecker hinzuzufügen, um Laufzeitfehler durch falsche Datentypen zu vermeiden. Ein Vorteil davon wäre, dass, direkt nach dem Parsen der Eingabe und vor der Instanziierung der Formel, das Programm mit einer Fehlermeldung beendet werden kann. In dieser Fehlermeldung sollte dann auch mit einer Zeilen- und Spaltennummer beschrieben werden, an welcher Position in der Eingabe das Problem aufgetreten ist.
Bisher lassen sich zudem nur reguläre Sprachen als Definitionsbereich für Wortparameter angeben. Eine Erweiterungsmöglichkeit wäre hier also, die Aufzählung von kontextfreien Sprachen hinzuzufügen. Die Definitionsbereiche können dann beispielweise mit einer kontextfreien Grammatik angegeben werden.

Literatur

[1] Maurice Herwig u. a. *Problem-Specific Visual Feedback in Discrete Modelling.* Proceedings of DELFI 2024. 2024. DOI: 10.18420/delfi2024_08.

[2] Norbert Hundeshagen, Martin Lange und Georg Siebert. "DiMo - Discrete Modelling Using Propositional Logic". In: *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings.* Hrsg. von Chu-Min Li und Felip Manyà. Bd. 12831. Lecture Notes in Computer Science. Springer, 2021, S. 242–250. DOI: 10.1007/978-3-030-80223-3_17. URL: https://doi.org/10.1007/978-3-030-80223-3_17.

[3] Norbert Hundeshagen. *Theoretische Informatik: Formale Sprachen und Logik. Vorlesung im Sommersemester 21.* Vorlesungsfolien. FG Theoretische Informatik/Formale Methoden, Universität Kassel, 2021.

[4] Dean N. Arden. "Delayed-logic and finite-state machines". In: *2nd Annual Symposium on Switching Circuit Theory and Logical Design (SWCT 1961).* 1961, S. 133–151. DOI: 10.1109/FOCS.1961.13.

[5] *DiMo Tool Web.* URL: https://dimofront.tools.tifm.cs.uni-kassel.de/dimo-tool (besucht am 23. 12. 2024).

[6] Maurice Herwig. *binaryCode.dm.* URL: https://cumbernauld.tifm.cs.uni-kassel.de/tools-for-students/dimo/DiMo/-/blob/2360be009bfa92c0563123ab1e34bd7a2691a52 examples/version_0.3/binaryCode.dm (besucht am 07. 01. 2025).

[7] Katrin Erk und Lutz Priese. *Theoretische Informatik. Eine umfassende Einführung.* Springer Berlin, Heidelberg, 2008. DOI: 10.1007/978-3-540-76320-8. URL: https://doi.org/10.1007/978-3-540-76320-8.

[8] M. DOUGLAS McILROY. "Enumerating the strings of regular languages". In: *Journal of Functional Programming* 14.5 (2004), S. 503–518. DOI: 10.1017/S0956796803004982.

[9] Egbert Harzheim. *Ordered Sets.* Springer New York, NY, 2005. DOI: 10.1007/b104891. URL: https://doi.org/10.1007/b104891.

[10] Xavier Leroy u. a. *The OCaml system release 4.08. Documentation and user's manual.* 6. Aug. 2019. URL: https://ocaml.org/manual/4.08/index.html (besucht am 08. 01. 2025).

[11] Vint Cerf. *ASCII format for network interchange.* RFC 20. Okt. 1969. DOI: 10.17487/RFC0020. URL: https://www.rfc-editor.org/info/rfc20.

[12] Vladimir I. Levenshtein. "Binary codes capable of correcting deletions, insertions, and reversals". In: *Soviet physics. Doklady* 10 (1965), S. 707–710. URL: https://api.semanticscholar.org/CorpusID:60827152.

[13] *Levenshtein distance - Wikipedia.* URL: https://en.wikipedia.org/w/index.php?title=Levenshtein_distance&oldid=1242756268#Definition (besucht am 06. 01. 2025).

[14] François Yergeau. *UTF-8, a transformation format of ISO 10646.* RFC 3629. Nov. 2003. DOI: 10.17487/RFC3629. URL: https://www.rfc-editor.org/info/rfc3629.

[15] Ganesh Gopalakrishnan. *Computation Engineering. Applied Automata Theory and Logic.* Springer New York, NY, 2006. DOI: 10.1007/0-387-32520-4. URL: https://doi.org/10.1007/0-387-32520-4.