Implementierung von Consize mittels eines Pattern-Matching-Systems

Diese Arbeit befasst sich mit der konkreten Implementierung der Programmiersprache Consize, auf Basis eines formalen Pattern-Matching-Systems, welches im Anhang B von Konkatenative Programmierung mit Consize beschrieben ist. Im ersten Teil des Berichtes wird ein kurzer Überblick über die Sprache und Implementierung von Consize gegeben. Es wird Argumentiert, dass sowohl die Implementierung von, als auch die Entwicklung mit konkatenativen Programmiersprachen, durch die Verwendung eines formalen Pattern-Matching-Systems deutlich vereinfacht werden kann.

Im ersten Abschnitt Was ist Consize — Ein sehr kurzer Überblick wird kurz Consize und dessen Funktionsweise beschrieben. Anschließend wird im Abschnitt Stapeleffekte: Notieren, Verstehen und richtig Interpretieren, garnicht so schwer oder. . . ? auf ein sehr menschliches Problem hingedeutet, dem die meisten konkatenativen Sprachen unterliegen. Danach wird in Eine Stapeleffekt-Notation für Menschen eine Pattern-Matching Notation als Lösung zu diesem Problem vorgestellt. In den Nachfolgenden Abschnitten 12 wird eine konkrete Implementierung zur maschinellen auswertung jener Notationen beschrieben; bevor in Consize mittels Pattern-Matching der derzeitige Fortschritt einer Consize Implementierung auf Basis des hier vorgestellten Systems wiedergibt. Nachdem ein Fazit und den möglichen zukünftigen Verbesserungen gezogen wird.

Was ist Consize — Ein sehr kurzer Überblick

Consize ist eine Programmiersprache, welche die funktionalle konkatenative Programmierung umsetzt. Anders als in gängigeren Programmiersprachen, werden in konkatenativen Programmiersprachen Funktionen nicht aufeinander angewandt, sondern konkateniert.

Dies führt zu einem völlig anderen Programmierstil, als wie jener von applikativen Programmiersprachen — wie etwar Java —, in welchen die Funktionsanwendung einer Funktion f auf einer Funktion g üblicherweiße wie folgt aussieht: $f(g(x))$. In Consize hingegen, sähe die selbe Funktionsanwendung folgendermaßen aus: xgf.

Im Prinzip funktioniert Consize auf einer Stapelverarbeitungsmachine (die Consize-VM), welche konzeptionell ein Callstack und einen Datastack umfasst. Beide Stapel enthalten sogenannte Wörter. Ein Wort kann ein Zeichen (`foobar`,`42`,`221B`), ein andere Stapel (`[ ]`,`[ 1 2 3 ]`) oder ein Wörterbücher `{ foo bar }` sein. Wörter, welche auf dem Datastack abgelegt sind, werden von der VM als reine Daten betrachtet, während Wörter, welche auf dem Callstack liegen von der Consize-VM interpretiert werden und ggf. einen definierten Effekt haben. Bspw. umfasst die Consize-VM einen vordefinieren Wortschatz von 56 primitiven Wörtern siehe Quellcode und deren (Stapel)-Effekte. Eines dieser Wörter wäre bspw. `dup`. Es sei angenommen, `dup` läge als oberstes Element auf dem Callstack, so wird die Consize-VM, das Wort `dup` im Wörterbuch der VM nachgeschlagen. Befindet sich zu `dup` ein Eintrag im Wörterbuch, wird der damit assoziierte Effekt eintreten — sofern dessen Vorbedingungen erfüllt sind. Im Falle von `dup` wäre der Stapeleffekt folgender: `(x -- x x)`, woebi linke Seite von `--` den Zustand des Datastacks vor, und die rechte Seite den Zustand nach der Ausführung von `dup` beschreibt. `(x -- x x)` bedeutet demnach soviel wie, "Nehme das oberste Wort vom Datastack und lege diesen, sowie ein duplikat des Wortes auf dem Datastack zurück". Die Vorbedingung wäre hierbei, dass sich zwangsweise ein Wort auf dem Datastack, vor der unmittelbaren Ausführung von `dup`, befinden muss.

Nun ist `dup` alleine nicht ausreichend, um ein vollständiges Programm zu beschreiben. Es wird ein Mechanismus benötigt, um eigene Wörter und deren Effekte zu kodieren. Bspw. wie es in anderen Programmiersprachen auch üblich ist, Funktionen zu konkatenieren, um komplexeres Verhalten zu beschreiben oder von diesem zu abstrahieren. Daher bietet die Consize-VM die Möglichkeit das interne Wörterbuch, um eigene/neue Wörter zu erweitern. Dazu werden diese Wörter aus der konkatenation der bestehenden Wörter gebildet — wie es auch unter gängigen Programmiersprachen mit Funktionen der Fall ist. Bspw. lässt sich das Wort `unpush` mit den Stapeleffekt `( [ itm & stk ] -- stk itm )` aus einer konkatenation folgender primitiver Wörter bilden: `dup pop swap top`.

Angenommen die obige Definition von `unpush` wäre im aktuellen Wörterbuch der Consize-VM vorhanden und `unpush` wäre das oberste Element auf dem Callstack. Dann würde die Consize-VM die Wörter `dup`,`pop`,`swap` und `top` auf den Callstack legen, so dass sie (wie hier angegeben) von links nach rechts einzeln Interpretiert und ausgeführt werden. Sprich zuerst würde `dup` ausgeführt werden, dann `pop` usw. Sollte die ausführung erfolgreich sein, bewirkt `unpush`, dass das oberste Element eines Stapels herausgenommen und als oberstes Element auf dem Datastack gelegt wird.

```
------------------------------------------------------------------------------
DS vor unpush:  ... [ Moriarty Sherlock Watson ]  | unpush ...  CS vor unpush
------------------------------------------------------------------------------
DS nach unpush: ... [ Sherlock Watson ] Moriarty  |        ...  CS nach unpush
------------------------------------------------------------------------------
```

Stapeleffekte: Notieren, Verstehen und richtig Interpretieren, garnicht so schwer oder...?

Anhand von `unpush` wird ein Problem deutlich. Um `unpush` zu verstehen, muss ein tiefgehendes Verständnis der Effekte von jeden einzelnen Worte, aus dem `unpush` zusammengesetzt ist, vorliegen. Andernfalls ist unklar, was `unpush` tatsächlich macht. Zwar deutet `unpushs` Stapeleffektdokumentation `( [ itm & stk ] -- stk itm )` darauf hin, was passieren sollte, jedoch ist diese Dokumentation nicht für alle Wörter gleich hilftreich. Betrachten wir die Definition von `each`.

```
: each ( seq quot -- ... )
    swap dup empty?
        [ 2drop ]
        [ unpush  -rot over [ call ] 2dip each ]
    if ;
```

Each erwartet auf dem Datastack eine Quotierung *quot* und darunter einen Stapel mit Wörtern *seq*. Für jedes Element aus *seq* wird `each` die Quotierung *quot* anwenden. Zur Veranschaulichung nehmen wir an, dass wir folgenden Datastack haben: `... [ 1 2 3 ] [ dup ]`, auf dem dessen `each` angewandt wird. Die Folge ist, dass `dup` auf `1`,`2` und `3` angewandt wird. Der Datastack nach `each` wäre entsprechend: `... 1 1 2 2 3 3`. Für Ungeübte wird die Definition von `each` schwer zu durchdringen sein. Die Stapeleffektdokumentation hilft hierbei auch nur bedingt — Wären Sie auch darauf gekommen, dass die Elemente nicht mehr in einem Stapel liegen? Ich hatte das anfangs nicht erwartet.

Diese Problem ist dem Entwickler von Consize bekannt. In der Dokumentation zu Consize Konkatenative Programmierung mit Consize wird daher im Anhang B eine alternative Notation für Stapeleffekte beschrieben, welche einerseits für Menschen verständlicher und andererseits von Maschinen auswertbar ist. Im nachfolgenden Abschnitt wird jene Notation beschrieben.

Eine Stapeleffekt-Notation für Menschen

Wenn wir uns die Anwendung eines hinreichend komplexen Wortes in Consize betrachten, wird klar, dass ein Word lediglich ersetzt wird, gegen dessen Definition. Welche wieder lediglich aus Wörtern besteht, die wiederum ersetzt werden gegen deren Definition usw. Solange, bis nur noch primitive Wörter vorhanden sind und

diese ausgewertet wurden. Demnach, entspricht die Auswertung eines Consize Programms einer Abfolge von Ersetzungsschritten; und damit die Consize-VM im Grunde einem Umschreibsystem.

Die in Anhang B von Konkatenative Programmierung mit Consize vorgeschlagene Stapeleffekt-Notation beschreibt genau jene einzelne Ersetzungsschritte, welche von der Consize-VM durchgeführt werden. D. h., wie sieht der aktuelle Zustand des Systems vor und nach einem Ersetzungsschritt aus. Genauer:

1. Wie sieht der Datastack zum Zeitpunkt vor dem Ersetzten eines Wortes aus und
2. wie sieht danach aus; Sowie,
3. Wie sieht der Callstack zum Zeitpunkt vor dem Ersetzten eines Wortes aus und
4. wie sieht der Callstach danach aus.

Betrachten wir wieder das Wort `dup`. In der Stapeleffekt-Notation vom Anhang B, wird `dup` wie folgt beschrieben.

```
#X | dup -> #X #X |
```

Wir nennen einen solchen Ausdruck Regel(-Beschreibung) und lesen wie folgt:

- Alles links von einem Pfeil (`->` oder `=>`), nennen wir Matching-Pattern (M-Pat).
- Alles recht von einem Pfeil (`->` oder `=>`), nennen wir Instantiation-Pattern (I-Pat).

Die Pattern setzten sich wiederum aus zwei Teilen zusammen:

1. Dem Datastack-Pattern, welches alles links von einem `|`-Symbol ist und
2. dem Callstack-Pattern, welches alles recht von einem `|`-Symbol.

In Consize können wiederum drei Verschiedene Dinge auf einem Stapel auftauchen: andere Stapel, Wörterbücher und Wörter. Diese werden wie folgt in der Notation ausgedrückt.

- Ein Stapel beginnt mit einer öffnenden `[` und muss mit einer `]` enden — auch hier gilt, dass innerhalb des Stapels wieder Stapel, Wörterbücher und Wörter erscheinen können.
- Ein Wörterbuch beginnt mit einer öffnenden `{` und muss mit einer `}` enden. Es gilt zu beachten, dass sich Wörterbücher von Stapel darin unterscheiden, dass deren Elemente immer Wort-Paare sein müssen, welche wiederum Stapel, Wörterbücher und Wörter sein können. Das heißt, Wörterbücher mit ungerader Anzahl an Elementen (wie etwar `{ a }` oder `{ a b c }`) sind nicht zulassig — wie in Consize auch.
- Alle anderen Zeichen sind Literale, mit Ausnahme von Worten, welche mit einem `#`-Symbol oder `@`-Symbol beginnen. Jene nennen wir Matcher. Bzw. Worte, welche mit einem `#`-Symbol beginnen, nennen wir Hash-Matcher; und Worte, welche mit einem `@`-Symbol beginnen, nennen wir AT-Matcher. Deren Semantik wird noch noch erläuter.

Zuerst muss festgehalten werden, dass die obige Regel beschreibung eine vereinfachte Darstellen (zur Verbesserung der Lesbarkeit) ist. Denn, in dieser werden lediglich Elemente auf den Stapeln angegeben, welche von einen Umschreibschritt betroffen sind. Das wäre einmal das oberste Elemente auf dem Datanstapel und Callstack. Es wird jedoch keine Aussage darüber getroffen, was mit allen anderen Werten auf den Stapeln passiert, noch welches Element das erste auf einem Stapel ist (ist das erste `#X`, oder das zweite auf dem Zielstapel, das oberste Element auf dem neuen Datastack?). Dies geschieht implizit. Denn sobald `->` verwendet wird, muss die Regel wie folgt gelesen werden.

```
@RDS #X | dup @RCS -> @RDS #X #X | @RCS
```

Die AT-Matcher `@RDS` und `@RCS` stehen führ alle restlichen Elemente auf dem Datastack bzw. Callstack. Nun können wir sehen, dass das oberste Element vom Datastack immer rechts steht, während das oberste Element vom Callstack immer links steht. Außerdem, sehen wir, was mit den anderen Elementen auf dem Stapeln passiert, nachdem die Regel angewandt wurde. Ein Beispiel. Angenommen wir hätten folgenden Datastack `1 2` und den Callstack `dup +`:

```
1 2 | dup +
```

Dann würde der Datastack und Callstack nach der Anwendung von `dup` folgendermaßen aussehen:

```
1 2 2 | +
```

Jetzt wird auch die Bedeutung der @- und #-Matcher deutlich. Ein #-Matcher ist im M-Pat ein Platzhalter, der für **ein** beliebiges Element stehen kann. Während ein @-Matcher eine beliebig lange Sequenz von Elementen zusammenfasst. Im obigen Beispiel stand `#X` für das Literal 2 und `@RDS` für alle restlichen Elemente des Datastacks, also `[ 1 ]`.

Das implizite Hinzufügen von @-Matchern macht die kurzform deutlich einfacher zu lesen, hat jedoch eine erhebliche Konsequenz. Regeln — wie bspw. `clear` — lassen sich nicht mit dieser Ausdrücken. Zur Verdeutlichung: In der vereinfachten Regelnotation würden wir `clear` so beschreiben: `| clear -> |`, was den Eindruck erweckt, dass `clear` den gesamten Datastack leert. In Wirklichkeit besagt die Regel jedoch, dass `clear` kein Element vom Datastack erwartet und auch nicht auf dem neuen Datastack veränder. Nach der Regeldefinition, wäre `clear` eine NOP, es macht garnichts. Der Grund hierfür ist das implizite Hinzufügen von `@RDS` und `@RCS`. Die Regel ist: `@RDS | clear @RCS -> @RDS | @RCS`.

Wollen wir Regeln wie das Verhalten von `clear` beschreiben können, muss `=>`, anstelle von `->` verwendet werden. Mit `=>` wird ausgedrückt, dass keine @-Matcher implizit hinzugefügt werden sollen. Damit wäre die korrekte Regel für `clear`: `@RDS | clear @RCS => | @RCS`.

Damit wären die wichtigsten Formalien zur Notation der Umschreibregeln beschrieben. Im folgenden Abschnitt wird darauf eingegangen, wie das

beschriebene Pattern-Matching-System in diesem Projekt zur Zeit implementiert ist.

Auswertung der Umschreibregeln — eine mögliche Implementierung in Python

Wenn wir Consize auf Basis der vorher eingeführten Regeln-Notation beschreiben und maschinell ausführen wollen, benötigen wir einen Interpreter der jene Regeln auswertet. Grundliegend muss dieser lediglich zwei Schritt wiederhohlt anwenden.

1. Finde eine anwendbare Regel und
2. wende jene Regel an.

Schritt 1. Regel Finden und Matching

Für Schritt 1. gibt es verschiedene Ansätze — siehe dazu den entsprechenden Abschnitt in Zukünftige Verbesserung für andere Ideen. In der vorliegenden Implementierung handelt es sich um einen sehr einfachen Ansatz. Zunächst gehen wir davon aus, dass dem Interpreter ausschließlich korrekte Regeln, in einer für ihn verarbeitbaren Form, vorliegen. Diese Regeln stehen in einem sog. Regelwerk, welches letztlich eine einfache Liste ist. Der Interpreter prüft, sequentiell, für jede Regel aus dem Regelwerk, ob diese Anwendbar ist. Ist dies nicht der Fall, prüft er die Anwendbarkeit der nächsten Regel. Sollte eine Regel anwendbar sein, wird er das Instantiation-Pattern (rechte Seite der Regel) der Regel umsetzten (Schritt 2.) und anschließend wieder vom Anfang des Regelwerks, alle Regeln durchprüfen. Dies wird solange wiederhohlt, bis keine Regel mehr anwendbar ist — was dem Programmende gleichkommt, weil kein Fortschritt mehr erziehlt werden kann. Die Implementierung dieser Schleife und dem Matching sind in den Methoden `make_step` und `run` in der *Interpreter.py* Datei kodiert.

Damit der Interpreter weiß, ob ein Pattern zutrifft, muss er den aktuellen Callstack und Datastack mit den Callstack bzw. Datastack vom M-Pat abgleichen. Auch hier gibt es wieder verschiedene Ansätze. Wir werden uns jetzt auf die aktuelle Implementierung fokusieren — für andere Ansätze siehe Abschnitt Alternative Pattern-Matching Strategien.

Das Pattern-Matching beginnt an oberster Stelle jeden Stapels und erfolgt rekursiv, bis das Pattern vollständig zutrifft, oder nicht. Trifft ein Pattern nicht zu, wird dies mit $False$ angegeben. Andernfalls, muss aus dem Pattern-Matching ein Zuordnung $Matches$ hervorgehen von allen Werten, die von einem #- oder @-Matcher gematcht wurden — wie wir das bereits im `dup` Beispiel gesehen haben. Der Algorithmus betrachtet das oberste Element von dem zu matchenden Stapel e_s und dem angegeben Pattern e_p. Nun gibt es fünf Fälle zu beachten.

- Literal: Ist e_p ein Literal, muss e_s das gleiche Literal sein. Trifft dies zu, wird mit den nächsten Elementen fortgefahren. Andernfalls, trifft das

Pattern nicht zu.

- Stapel: Ist e_p ein Stapel, muss e_s auch ein Stapel sein. Trifft dies zu, werden die Stapel miteinander gematcht. Dies geschieht rekursiv. Matcht der Stapel dem gegeben Stapelpattern, wird mit den nächsten Elementen fortgefahren.
- Wörterbuch: Ist e_p ein Wörterbuch, muss e_s auch ein Wörterbuch sein. Trifft dies zu, werden beide Wörterbücher miteinander gematcht. Dies geschieht ebenfalls rekursiv. Hier sei angemerkt, dass die aktuelle Implementierung jedes Wörterbuch als eine Liste betrachtet. Dies ist zwar ineffizient, bietet allerdings die Möglichkeit, dass die Notation und Match-Logik zwischen Wörterbücher und Stapeln nicht unterscheiden.
- #-Matcher: Ist e_p ein #-Matcher, wird e_p mit e_s assoziiert. D. h. sie werden in $Matches$ abgelegt, sofern für e_p noch keine Zuordnung in $Matches$ existier. Existiert bereits eine Zurodnung, muss e_s dem bereits zugeordneten Wert gleichen. Andernfalls, trifft das Pattern nicht zu.
- @-Matcher: Ist e_p ein @-Matcher, werden die Lesereihenfolge der Elemente im Pattern, sowie dem zu matchenden Stapel umgekehrt und das matching weiter forgeführt. Bis wieder zum @-Matcher angekommen wurde, woraufhin die verbleibenden Elemente dem @-Matcher zugeordnet werden.

Der @-Matcher ist etwas kompliziert. Die Implementierung ist so umgesetzt, weil bspw. folgendes Pattern erlaubt sein: `#LAST 2 @MID #FIRST`. Hier würde `@MID` alle Elemente zwischen dem ersten und letzten Element eines Stapels zugeordnet bekommen. Wenn der @-Matcher einfach den kompletten Rest eines Stapels matchen würde, wäre er Greedy und der obige Ausdruck nicht mehr möglich. Nutzten wir das Pattern auch nochmal, um die Implementierung des Algorhtmus genauer zu verdeutlichen. Dazu nehmen wir an, dass das Pattern auf den Datastack `1 2 3 4` gematcht wird. Der Algorithmus wird zuerst `#FIRST` als e_p zum matchen wählen, weil bei Datastack das erste Element rechts steht. Er findet auf den zu matchenden Stapel die `4` und ordnet damit `#FIRST` dem Element `4` zu und entfernt diesen vom Stapel. Nun folgt der @-Matcher `@MID`. Zuerst wird geprüft, ob bereits eine Zuordnung für `@MID` existiert, ist dies der Fall, beenden wird das Pattern-Matching und geben alle Zurodnungen/matches zurück. Andernfalls, wird diesem zunächst eine Referenz auf dem zu matchenden Stapel zugeordnet: Also `@MID=[ 1 2 3 ]`. Nun wird die Leserichtung gewechselt, auf links-nach-rechts. D. h. der nächste Matcher ist `#END`. Diesem wird das Element `1` zugewiesen und `1` anschließend vom Stack runtergenommen. Weil, `@MID` lediglich eine Referenz hat, sieht die zuordnung nun folgendermaßen aus: `@MID=[ 2 3 ]`. Im nächsten Schritt wird das Literal `2` gematcht; welches mit dem Literal `2` auf dem Datastack matcht. Die Folge: Beide Literale werden vom Stapel runtergenommen, womit sich `@MID` erneut aktualisiert auf `@MID=[ 3 ]`. Nun sind wir erneut bei `@MID` angekommen. Da bereits eine Zurodnung für `@MID` existiert, wird hier abgebrochen und alle Zuordnungen zurückgeliefert. — Die Implementierung des Matching Algorithmus befindet sich in der Datei *StackPattern.py*, Methode `match`.

Beachte: Die derzeitige Implementierung setzt voraus, dass es niemals mehr als

nur einen @-Matcher innerhalb eines M-Pat gibt. Andernfalls wären folgende Ausdrücke möglich: `[ @LEFT 3 @RIGHT ]`, welche nicht zulässig sind, weil es mehrere Lösungen gibt. So könnte bei einem Datastack mit den Elementen `1 3 3 3 7`, `@LEFT` die Literale `1 3 3` oder `1 3` oder nur `1` zugeordnet bekommen. Das Matching soll aber immer nur eine eindeutige Lösung erzeugen.

Ist das Pattern-Matching erfolgt sowohl für den Callstack und den Datastack. Ergeben beide matching vorgänge ein Zuordnung (also nicht *False*), kann die Regel angewandt werden.

Schritt 2. Regel Anwendung bzw. Instanziierung

Die Regelanwendung ist trivial. Für beide Instantiation-Pattern (Datastack und Callstack) werden die ie Instanziierung vorgenommen und deren Ergebnis sind der neue Data- bzw. Callstack. Dies erfolgt für ein Pattern wie folgt: Gehe über das Instanziierungs-Pattern und ersetzte jeden Matcher mit den Werten, welche diesem zugeordnet wurde. Fertig. — Die Implementierung des Instantiation Algorithmus befindet sich ebenfalls in der Datei *StackPattern.py*.

Nach der Instanziierung und dem setzten der neuen Stacks, wird der Interpreter — wie bereits beschrieben — erneut versuchen eine Regel zu finden, welcher er anwenden kann. Damit wäre die grundlegende Funktion des Interpreters abgedeckt. Im nachfolgenden Abschnitt wird noch ein wichtiges Problem von dem Pattern-Matching-System erläutert, welches gelöst werden muss, bevor Consize überhaupt auf dem diesem Aufgebaut werden kann.

Plugin System für native Wörter

Die Umschreibregeln sind lediglich für die Beschreibung von Stapeleffekten geeignet. Für ein sinnvolles Programm benötigt es jedoch mehr. So möchte man vlt. ein Berechnungsergebnis auf der Konsole ausgeben. Mit den Umschreibregeln ist dies nicht möglich. Es bedarf einer Schnittstelle, mit der das Pattern-Matching-System mit der Umwelt kommunizieren kann. In der Consize-VM erfolgt diese kommunikation über fest einprogrammierte Wörtern; wie etwar `slurp`, was einen Dateipfad auf dem Datastack erwartet und bei seiner Anwendung, die Datei einliest und deren Inhalt auf dem Datastack legt. Das selbige Prinzip ist auch in der vorliegenden Implementierung umgesetzt, allerdings noch etwas erweitert.

Anstelle diese nativen Wörter fest im Regelwerk einzukodieren, werden diese über ein Pluginsystem, zur Laufzeit, geladen. Das hat den Vorteil, dass Wörter welche sich nicht über Umschreibregeln ausdrücken lassen auch nachträglich, ohne Anpassung des Interpreters, hinzufügen lassen. Das mag für Python keine Rolle spielen, weil es sich um eine interpretierte Sprache handelt, das Prinzip lässt sich aber auch in einer kompilierten Sprache umsetzten und dann muss der Interpreter nicht mehr erneut gebaut werden. Nachfolgend wird das laden jener nativen Wörtern beschrieben.

Der Interpreter sucht in einem vordefinierten Verzeichnis *native-words* (konfigurierbar über Kommandozeilenargument) nach Python-Module und lädt diese — der Quellcode hierzu befindet sich in *Interpreter.py*, Methode `discover_native_rules`. Das sind im Prinzip Python-Quellcode-Dateien. Beim Laden werden diese Interpretiert und deren Inhalt, der aktuellen Programmausführung zur verfügung gestellt. In diesen Python-Modulen sollten sich idealerweiße die Definitionen der nativen Wörter befinden.

Jede Definition eines nativen Wortes muss natürlich einer, vom Interpreter vorgegebenen Schnittstellen entsprechen. Diese Schnittstelle ist die Klasse *NativeRule*, von der jedes native Wort letztlich abgeleitet sein muss, damit der Interpreter das native Wort finden, instanziieren und damit arbeiten kann. Das geht, indem man sich von Python alle Klassen geben lässt, welche von einer anderen Klasse abgeleitet sind — `discover_native_rules`. Liegen Subklassen von *NativeRule* vor, wird der Interpreter eine Instanz von jeder dieser Klassen erzeugen und in seinem Regelwerk aufnehmen, womit sie schließlich verwendbar werden.

Instanzen der Klasse *NativeRule*, sind ebenfalls Unterklassen von *IRule*, welche allesamt das Command-Pattern umsetzten. Sprich, jedes Wort hat eine `execute`-Methode, welche der Interpreter für jede Regel, für den Matching- und Instanziierenungsschritt, mit sich als Argument, aufruft. Über die übergebene Interpreter-Referenz an `execute`, hat jede Regel letztlich Zugriff auf den aktuellen Call- und Datastack. Wird `execute` vom Interpreter aufgerufen, prüft jede Regel für sich selbst, ob diese anwendbar ist. Ist dies der Fall, wird sie den Data- bzw. Callstack entsprechend ändern. Der Interpreter selbst ist somit völlig von der Implementierung der einzelnen Regeln entkoppelt. Sehr Vorteilhaft ist dies, wenn für jede Regel noch die Dokumentation mitgeliefert wird. Hier kann der Interpreter eine Hilfe selbstständig ausgeben — siehe die Datei *native-words/00-interpreter-commands.py*.

Mit der Unterstüzung von nativen Worten steht einer Implementierung von Consize auf dem Pattern-Matchting-System nichts mehr im Wege. Im folgenden Abschnitt wird der aktuelle der Implementierung beschrieben.

Implementierung von Consize

Eines sei vorweggenommen. Consize ist in seiner vollständigkeit noch nicht lauffähig. Die Grundlegenden primitiven Wörter, welche sich mit Umschreibregeln ausdrücken lassen, wurden alle in dieser ausgedrückt. Zufinden sind diese in der Datei *consize.ruleset*. Außerdem wurden Regeln für jene Wörter ergänzt, welche mit Wörterbüchern aggieren. Weiterhin wurden die nativen Wörter der Consize-VM imlementiert. Damit sollte ein minimale Arbeiten mit den gänigsten Operationen möglich sein. Die korrekte Funktion der Wörter sind von entsprechenden Unit-Tests abgedeckt.

Zusätzlich wurde der Versuch unternommen, die Prelude zu laden, um die

Consize REPL zu betreten. Es scheint jedoch ein Problem mit dem laden des bootimages vorzuliegen. Weswegen Wortdefinitionen in folgender Form `: unpush dup pop swap top ;` nicht möglich sind, was wiederum das laden der Prelude verhindert. Schließlich sind alle Definitionen in der Prelude über Form definiert. Der Interpreter selbst unterstützt jedoch das hinzufügen von neuen Regeln über die native Implementierung von `def` — siehe Ende von *consize-words.py*. Es können also Wort Konkatenationen/Redefinitionen vorgenommen werden.

Es ist bis dato (10. August 2024) unklar, warum die in bootimage definierten Wörter — insbesondere `:` und `scanf` nicht korrekt funktionieren. Es wurden explizit Wort-Definitionen herausgenommen, welche bereits mit den Umschreibregeln umgesetzt wurden. So, dass beim laden des bootimages bspw. nicht `def` mit der alten Implementierung überschrieben wird, welches wegen dem Fehlen von `get-dict` und `set-dict` in der neuen Implementierung, garnicht funktionieren kann. Außerdem wurden die letzten Worte des bootimages `mapping get-dict merge set-dict` entfernt. Das bootimage ließ sich letztlich erfolgreich laden, so dass die darin definierten Wörter im Regelwerk des Interpreters standen. Deren Aufruf allerdings nicht das gewünsche Ergebnis erzielten.

Fazit

Leider konnte eine vollständige Consize Umgebung nicht erfolgreich umgesetzt werden. Jedoch ist mit dem funktionierenden Pattern-Matching-System und dem umgesetzten Plugin-System ein solides Fundament für eine zukünftige Umsetztung und Weiterentwicklung gesichert. Einige Verbesserungen oder Anregungen sind noch im nachfolgenden Abschnitt beschrieben.

Zukünftige Verbesserungen

Zur Consize Umsetztung

Zunächst wäre es schön eine funktionierende Consize REPL zu haben. Dann könnte das Rechnen, welches zur Zeit mit nativen Wörtern implementiert ist, auf Umschreibregeln umgesetzt werden. Das ist zwar langsam, aber das Rechnen mit Umschreibregeln wäre ein schöne Demonstration von formalen Systemen. Außerdem können noch einige Wörter aus der Prelude in den Umschreibregeln umformuliert werden.

Pretty Print Words

Was bisher nicht erwähnt wurde ist, dass der Interpreter die einzelnen Ersetzungsschritte als eine *Chain of Reasoning* — siehe Anhang B von Konkatenative Programmierung mit Consize — ausgeben kann. Diese Ausgabe verwendet bereits Farbkodierung, um Wörter auf dem Callstack, oder den Stacktrenner, farblich hervorzuheben. Interessant wäre es, wenn Wörter mit einem Unterstrich

markiert wären. Daran könnte schnell gesehen werden, ob auf dem Stack mehrere einzelne Wörter liegen oder ein Wort, welches whitespace enthält. Bei letzterem würde der whitespace auch unterstrichen werden.

Validierung und Analyse von Regeln

Momentan gibt es keine semantische Validierung. Somit können Regeln formuliert werden, welche nicht erlaubt sind. Bspw. dürfen nicht mehrere @-Matcher in einem Pattern vorkommen. Mit einen Validierungsschritt könnten solche Regeln ausgeschlossen werden.

Weiterhin könnte eine erweiterte Analyse durchgeführt werden, ob bestimmte Regeln jemals anwendbar sind oder andere überschatten. Hilfreich könnte hierfür das Specification Pattern sein, wie es hier von Eric Evans und Martin Fowler beschrieben ist.

Lastreduktion beim finden von Regeln anwendbaren Regeln

Anstelle jede Regel einzeln auf ihre Anwendbarkeit zu prüfen, können viele Regeln bereits vorzeitig ausgeschlossen werden, indem zunächst ausschließlich der Callstack betrachtet wird. D. h., wenn das Wort `dup` auf dem Callstack liegt, müssen auch nur die Regeln geprüft werden, welche `dup` auf dem CS-Pattern hat.

Eine HashMap könnte das Wort als Key verwenden und als Value eine Liste von Regeln, die jenes Wort als oberstes Element auf dem Callstack haben. Dafür müssten alle Regeln so umgeschrieben werden, dass jede Regel nur ein Element auf dem Callstack im M-Pattern aufweißt. Diese Umformung sollte vom Interpreter intern vorgenommen werden, sodass der Nutzer nicht eingeschränkt wird. Für das Escape-Word `\`, dessen CS-Pattern `\ #H` ist, kann evtl. keine Umformung gefunden werden — hierfür müsste ich nochmal nachdenken.

Es könnten auch alle Wörter, welche auf dem Callstack im Instantiation-Pattern vorkommen zu Funktionen transformiert werden. Diese Funktionen sind dann keine Regeln mehr die nachgeschlagen werden müssen, sondern einfach Funktionen die vom Interpreter direkt aufgerufen werden. Dabei sollte beachtet werden, dass eine solche Funktion nicht ihre Subfunktionen selbst aufruft, da sonst der Callstack u. U. nicht ausreichen wird. Außer die Aufruftiefe wird begrenzt, indem Funktionen in Packete gepackt werden, welche Garantieren, dass eine bestimmte Aufrufstiefe niemals erreicht wird. Oder einfacher, jede Funktion legt ihre Unterfunktionen auf den Callstack des Interpreters und dieser führt sie nacheinander durch. Somit ist immereine Aufrufstiefe von 1 gegeben. Das ist letztlich das Prinzip, nachdem die Wortkonkatenationen im aktuellen System funktionieren.

Alternative Pattern-Matching Strategien

Im Abschnit Regel finden und Matching wurde die Implementierung des Pattern-Matching-Algorithmus beschrieben. Dieser könnte auch anders funktioneren. Eine weitere prototypische Implementierung liegt bereits in der Datei *playground/ultra-small-matching-function.py* vor. Deren Ansatz hier noch kurz beschrieben werden soll.

Pattern-Matching mit Python unpack-Operationen

Python unterstützt das unpacken von Listen mit einer sog. unpack-expression, welche sehr stark der Pattern-Matching Notation von dem hier beschriebenen System ähnelt. Steht bei einer Zuweisung in Python auf der rechten Seite eine Liste, können auf der linken Seite mehrere Variablen angegeben werden. Die Variablen erhalten dann die einzelnen Werte der Liste genauso wie sie bei unserer Pattern-Notation mit #-Matcher beschrieben wurde. Auch der @-Matcher ist vertreten, indem vor einer Variable `*` angegeben wird. Die Idee: Nehme unsere die Pattern-Beschreibung und schreibe diese so um, dass sie aussieht wie eine Python unpack-expression. Bsp. aus dem Pattern `#F [ 2 @GREETING [ #NUM ] 4 ] #L` wird folgender String: `H_F, ( _2, *AT_GREETING, ( H_NUM ,) _4 ,), H_L`. Diesen kann man über `exec` Python interpretieren lassen exec(f"{pattern} = {stk}“)‘. Anschließend lässt man sich alle Variablen geben und schreibt deren Zugewiesene Werte mit den Ursprünglichen Namen zurück in eine Map. Die Implementierung ist deutlich kürzer und leicher verständlich. Der Nachteil ist jedoch, sie ist stark Python abhängig und nicht in andere Sprachen überführbar. Deswegen ist dies derzeit nicht die Standardimplementierung. Außerdem wurde sie noch nicht ausreichen genug getestet. Es wäre jedoch Interessant zu sehen, ob diese Implementierung genauso robust und evtl. schneller ist als die derzeitige.

Pattern-Matching via Reguläre Ausdrücke

Bisher habe ich nur gehört, dass sich das hier beschriebene Pattern-Matching-System nicht mit regulären Ausdrücken umsetzten lässt. Ein schwerwiegendes Argument ist, dass die balansierte Klammerung nicht mit einem regulären Ausdruck beschrieben werden kann. Ich bin allerdings auch RegEx-Engines gestoßen, welche rekursive Patternaufrufe unterstützten. Diese werden u.a. dafür verwendet, um balansierte Klammer auszudrücken. Es wäre sehr Interessant zu erfahren, ob das System mit solch einer Engine nicht doch umsetzbar wäre.

Quellen:

- Konkatenative Programmierung mit Consize