Sofun -
Konzept und Implementierung einer stackorientiert-funktionalen Programmiersprache

Henri Schellberg

25.3.2018

Inhaltsverzeichnis

1 Spezifikation

1.1 Das stackorientiert-funktionale Paradigma

Sofun (Abk.: Stack Oriented FUNctional language) ist eine stackorientierte Sprache: Alle Ausdrücke (d.h. Werte und Funktionen) werden vom Interpreter in Stacks aufgelöst und als Datenstruktur stehen nur Stacks zur Verfügung. Funktionen können dabei als - teilweise argumentabhängige - Repräsentationen von Abschnitten des aufrufenden Stacks angesehen werden: Sie werden vom Interpreter, wie Inline-Funktionen in imperativen Sprachen, durch die repräsentierten Stacks ersetzt. Nach dem Parsing besteht also das ganze Programm aus einem einzigen Stack aus Elementarfunktionen - dem Mainstack.[1] Dieser wird evaluiert und ergibt den Rückgabewert des Programms. Bis zur Evaluation kann man also Ähnlichkeiten mit der Ableitung einer Zeichenkette in einer linksregulären Grammatik erkennen, wobei die Funktionsaufrufe als Nichtterminale, Funktionsdefinitionen als Produktionsregeln und Elementarfunktionen und ganze Zahlen als Terminale fungieren: Ein Vergleich, der sich allerdings durch die Komplexität von Funktionsdefinitionen als unvollständig erweist.

Die Sprache teilt außerdem einige Eigenschaften einer funktionalen Sprache, als strikt funktionale Sprache kann sie aber nicht bezeichnet werden. Warum das der Fall ist, wird nach der syntaktischen und semantischen Spezifikation noch genauer behandelt werden.

1.2 Die Syntax

Sofuns Syntax lässt sich mit folgender formaler Grammatik beschreiben:

1.2.1 Die Terminale

number [$\in \mathbb{N}$]
(
)
:
?
string [nicht rein numerische Zeichenkette]
ϵ [leere Zeichenkette]

[1] Das ist, wie unter *Implementierung* genauer erklärt, in der Praxis nicht vollständig richtig, weil die Auswertung der Elementarfunktionen und der definierten Funktionen in einem Schritt passiert, aber es kann als Gedankenmodell davon ausgegangen werden.

1.2.2 Die Nichtterminale und ihre Produktionsregeln

FUNCTION → HEAD : STACK || HEAD ? BODY
HEAD → name HEAD || ϵ
BODY → STACK : STACK ? BODY || STACK
DATASTACK → (STACK)
STACK → name STACK || number STACK || DATASTACK STACK || ϵ

1.3 Die Semantik

Sofunprogramme bestehen aus mehreren verknüpften Stacks, die durch Parsing schließlich zu einem einzigen Mainstack verschmelzen. Dieser wird evaluiert und gibt seinen Rückgabewert an den Nutzer zurück. Eine Funktion ist ein Bezeichner für einen Stack. Es ist dabei wichtig zu verstehen, dass der Funktionskörper nicht evaluiert wird: die Funktion wird im aufrufenden Stack einfach durch ihren Körper ersetzt. Obwohl also Funktionen in Sofun im funktionalen Sinne keinen Rückgabewert besitzen, werde ich der Einfachheit halber im Folgenden von dem Abschnitt des Mainstacks, der anstelle des Funktionsaufrufs getreten ist, als Rückgabestack sprechen.
Programme sind aber nicht nur auf Interpreterebene als Stacks repräsentiert, sie werden auch als solche geschrieben:

1.3.1 Reverse Polish Notation

Reverse Polish Notation, kurz RPN, ist eine ursprünglich für Taschenrechner entwickelte Variante der Notation, bei der Funktionen und ihre Argumente als von Leerzeichen getrennter Stack hintereinander geschrieben werden. Dabei kann man vollständig auf ordnende Klammern verzichten: `2 + 3` in der normalerweise üblichen *Infix-Notation* wird in RPN nur zu `2 3 +` umgeordnet, was aber in Infix `(3 + 4) * (5 - 2)` wäre, ist in RPN ganz klammerlos `3 4 + 5 2 - *`. Zur Evaluation dieses Ausdrucks wird zuerst die linkeste Funktion `+` mit ihren Argumenten `3 4` aufgerufen. Das ergibt 7. Dasselbe geschieht mit `5 2 -`: 3. Nun ist der Stack: `7 3 *`. Sieben mal drei ist 21. So erhalten wir dasselbe Ergebnis wie in der korrespondierenden Infix-Notation.

1.3.2 Funktionen

Funktionen bestehen in Sofun aus einem Funktionskopf, der den Funktionsnamen und die Argumente beinhaltet, und einem Funktionskörper, der den Rückgabestack und gegebenenfalls eine Verzweigung beinhaltet. Kopf und Körper sind durch einen Doppelpunkt getrennt. Unverzweigte Funktionskörper stellen einen normalen Stack aus Funktionen, Zahlen, Datastacks und den Argumenten der Funktion dar. Verzweigte Funktionskörper besitzen mehrere Zweige, die durch Fragezeichen getrennt sind. Jeder einzelne dieser Zwei-

ge besteht aus einer Kondition, und, wieder getrennt mit einem Doppelpunkt, dem im Fall ihrer Erfüllung gültigen Rückgabestack. Es gibt also verschiedene Komplexitätsgrade von Funktionen:

Der Name ist der einfachste Typ von Funktion: `a : 1 2 3` gibt `1 2 3` zurück. Hier wird einfach ein Symbol als Bezeichner für mehrere andere Symbole festgelegt, der Rückgabestack steht vor Aufruf fest.

Die argumentabhängige Funktion ist der nächst komplexere Funktionstyp: `a b first : a` gibt das erste übergebene Argument zurück, repräsentiert also abhängig davon, welche Argumente bei Aufruf übergeben werden, unterschiedliche Rückgabestacks.

Die verzweigte Funktion ist der komplexeste Typ: `a b <= ? a b < : 1 ? a b = : 1 ? 0` besteht aus mehreren Rückgabestacks, die von Konditionen abhängig sind. Nach jedem Fragezeichen steht eine Kondition. Diese wird evaluiert. Gibt sie keine 0 und keinen leeren Stack zurück, wird der nach dem Doppelpunkt stehende Körper zurückgegeben. Trifft dies nicht zu, so wird die nächste Kondition geprüft. Hinter dem letzten Fragezeichen darf keine Kondition stehen, nur ein Rückgabestack, der zurückgegeben wird, wenn keine der vorherigen Konditionen zutrifft. In diesem Fall - der Definition von kleiner gleich aus der Standardbibliothek - wird also `1` zurückgegeben, falls a kleiner als b ist, ebenfalls `1` zurückgegeben, falls a gleich b ist und in jedem anderen Fall `0`.

Die Auswertung einer Funktion geschieht in folgenden Schritten: Zuerst werden Argumentsymbole im Funktionskörper durch die übergebenen Werte ersetzt, daraufhin werden die Konditionen ausgewertet und so der korrekte Rückgabestack identifiziert. Schließlich wird dieser zurückgegeben. Bei gleichen Argumenten repräsentiert eine Funktion also immer denselben Stack(-abschnitt).

1.3.3 Datastacks

Datastacks sind Stacks im Stack. Statt in den Mainstack eingegliedert, und damit evaluiert zu werden, werden sie vom Mainstack als ein einziges Element behandelt: `( 1 2 3 ) ( 4 5 6 ) first` gibt `( 1 2 3 )` zurück. Auch Funktionen werden so in einem Datastack nicht aufgerufen, was `( + ) ( 4 5 6 ) first` zu einem erlaubten Ausdruck macht, obwohl `+` im Datastack selbst nicht genug Argumente vorfände.
Auf Datastacks können alle üblichen Stackoperationen angewendet werden: `2 3 ( 1 + ) pop` gibt `2 3 +`, also `5` zurück.

1.4 Warum ist Sofun nicht funktional?

Wikipedia führt fünf Eigenschaften funktionaler Sprachen auf, an zwei von diesen lässt sich besonders gut erkennen, warum Sofun keine rein funktionale Sprache ist:

> Computerprogramme werden als Funktionen verstanden, die für eine Eingabe eine Ausgabe liefern, die nur von dieser abhängig ist.[2]

Zwar ist in Sofun tatsächlich die Ausgabe nur von der Eingabe abhängig, aber das Programm ist dennoch keine Funktion, denn Funktionen in Sofun geben nicht Ergebnisse zurück, sondern nur wieder unevaluierte Stacks. Ein Programm verhält sich anders, denn der Mainstack wird schließlich evaluiert und erst das Ergebnis dieser Evaluation wird vom Programm zurückgegeben. Ein Programm ist also mehr als eine Funktion.

> Funktionen sind gegenüber allen anderen Datenobjekten gleichberechtigt. Das bedeutet, dass sie als Parameter in Funktionen eingehen dürfen und ebenso als Berechnungsergebnisse aus Funktionen hervorgehen können.[3]

Funktionen sind in Sofun keine gleichberechtigten Datenobjekte: Sie sind nur Bezeichner für Stacks, sie existieren zur Laufzeit gar nicht wirklich. Sie lassen sich auch nicht als Argumente übergeben - nicht ohne sie in Datastacks zu kleiden, um sie vor Evaluation zu bewahren.

Trotz dieser Einwände lässt sich Sofun durchaus als funktional bezeichnen: die Sprache teilt in der Anwendung die meisten Konzepte einer funktionalen Sprache. Dem Programmierer werden die Unterschiede nur an wenigen Stellen tatsächlich auffallen.

2 Implementierung

Eine REPL für Sofun wurde von mir in C++ umgesetzt. REPL bedeutet Read-Evaluate-Print-Loop und beschreibt eine interaktive Konsole, in der Blöcke von Code direkt eingegeben, ausgewertet und einzeln debuggt werden können. Für funktionale Sprachen bieten sich solche REPLs besonders an, da die einzelnen Elemente, das heißt Funktionen, auch für sich genommen sinnvolle Ergebnisse liefern und aufgrund ihrer Kontextfreiheit in der REPL dieselben Ergebnisse liefern wie eingefasst in ein größeres Programm. Wie oben gezeigt, teilt Sofun diese Eigenschaften mit strikt funktionalen Sprachen, weswegen mir auch hier eine REPL geeignet erschien.

[2] *Wikipedia: Funktionale Programmierung.*
[3] *Wikipedia: Funktionale Programmierung.*

Meine Implementierung beinhaltet außerdem einen Interpreter, mit dem Programme direkt aus einer Datei ausgeführt werden können. Dabei evaluiert der Interpreter die Funktion namens *main* in der Datei, übergibt ihr die Commandline-Argumente und printet ihren Rückgabewert: `./sfrepl bar1 bar2 foo.fun`. Die Mainfunktion kann dabei natürlich andere in der Datei zuvor definierte Funktionen nutzen.

2.1 Die Nutzung der REPL

Die Sofunrepl, kurz sfrepl, interpretiert Einzeiler des Nutzers. Sie verfügt dabei aber über einen Speicher definierter Funktionen, wodurch auch längere Programme vollständig in ihr verfasst werden können. Stacks ohne Funktionskopf werden dabei als Mainstacks sofort evaluiert. Am Anfang jeder Sitzung wird die Standardbibliothek geladen, die sich unter dem Namen *std.fun* am Speicherort der REPL befinden muss. Weitere Dateien mit Funktionen können mit dem Befehl `:l`, gefolgt vom Dateinamen (erwünschte Dateiendung: .fun), geladen werden. Im Repository befindet sich beispielsweise die Datei *examples.fun*, deren Beispielfunktionen nach dem Laden in die REPL genutzt werden können. Sauber verlassen lässt sich die REPL mit `:q`. Mit `:d` lassen sich sehr ausführliche Debugnachrichten anzeigen, die allerdings eher zum Debuggen der REPL als zum Debuggen der selbstgeschriebenen Funktionen geeignet sind.

2.2 Der Aufbau der REPL

Die Sofunrepl ist in C++ verfasst. Stacks sind dabei als Vector<String>, also als in seiner Größe veränderlicher Array von Strings repräsentiert. Die Reallokation des Speichers muss dabei nicht händisch erledigt werden, sondern geschieht im Hintergrund automatisch. Es handelt sich hier also trotz ähnlicher Eigenschaften nicht um verkettete Listen. Vektoren haben viele Vorteile, allerdings auch Nachteile: Häufiges Verändern ihrer Größe ist sehr ineffizient. Das lässt sich besonders beim Umgang der REPL mit Datastacks spüren. Dieser Nachteil wird dadurch potenziert, dass C++-Strings sozusagen Vektoren von Chars sind, das Problem besteht so auf zwei Ebenen.

Aufgrund der mir zur Verfügung stehenden Zeit und auch der Zielsetzung des Projektes, nicht etwa eine besonders praktikable, sondern hauptsächlich eine im Aufbau neuartige und interessante Sprache zu entwickeln und dabei ihre Möglichkeiten und Grenzen zu entdecken, habe ich mich auch gegen die Implementierung eines Typsystems entschieden: Die Strings, die die einzelnen Elemente der Stacks ausmachen, sind vom Typ her undefiniert, die Built-In-Funktionen lassen sich allerdings bisher nur auf natürliche Zahlen und Datastacks anwenden.

Das C++-Programm besteht dabei, neben internen Helferfunktionen, nur aus diesen *Built-In-Funktionen*, sozusagen den Elementarfunktionen, die sich in der Sprache selbst nicht weiter durch einen Rückgabestack ersetzen lassen (z.B. +, pop, usw.), und drei Kernfunktionen:

parse parsed die Eingabe des Nutzers, macht also aus dem Input-String zuerst durch Spaltung an den Leerzeichen einen Vector<String> und ermittelt dann, ob es sich hier um eine Funktionsdefinition oder einen zu evaluierenden Stack handelt. Trifft ersteres zu, wird die Funktion unter ihrem Namen in eine C++-map (also einen Lookup-Table) eingetragen. Im zweiten Fall wird der betreffende Stack an die Funktion *evaluate* übergeben.

evaluate evaluiert einen Stack, das heißt, sie ersetzt alle darin enthaltenen Funktionsaufrufe durch ihre Rückgabestacks, indem für sie *parse_function* aufgerufen wird. Dasselbe geschieht mit allen Built-In-Funktionen, indem die entsprechenden C++-Funktionen aufgerufen werden. Dies geschieht solange, bis sich kein Funktionsaufruf mehr auf dem Stack befindet. Diesen evaluierten Stack gibt sie zurück.

parse_function ermittelt zu einem Funktionskopf mit übergebenen Argumenten den passenden Rückgabestack, dazu ersetzt sie die Argumentsymbole durch ihre tatsächlichen Werte, evaluiert gegebenenfalls die Bedingungen - wiederum mit *evaluate* - und gibt schließlich den richtigen Rückgabestack zurück.

3 Ein Beispiel: Quicksort

In der Datei *examples.fun* sind einige Codebeispiele aufgeführt. Eines davon will ich hier zum besseren Verständnis der Sprache kleinteilig erklären: Den Sortieralgorithmus „Quicksort“.
In Quicksort wird eine Liste (hier: ein Stack) sortiert, indem er nach der Größe der Elemente in Teilstacks aufgespalten wird, die wiederum in Teilstacks aufgespalten werden, die dann schließlich sortiert wieder zusammengefügt werden. Dieses Beispiel ist also geeignet, um die komplexeren Features von Sofun vorzuführen: Datastacks und Rekursion. Auf folgende Weise lässt sich Quicksort in Sofun implementieren:

```
a lesser : a popped ( a pop <= ) filter
a greater : a popped ( a pop > ) filter
a sort ? a size 1 <= : a ? a lesser sort a pop push a greater sort concat
```

`lesser` und `greater` sind dabei Helferfunktionen: Gibt man ihnen einen Stack, gibt `lesser` einen Stack aller Elemente zurück, die kleiner gleich dem obersten Element sind und `greater` einen Stack derjenigen, die größer als das oberste Element sind. Das geschieht

mithilfe eines Filters, einer Funktion aus der Standardbibliothek: Gibt man `filter` einen Stack und eine Bedingung, gibt `filter` einen Stack mit allen Elementen zurück, auf die die Bedingung zutrifft. Hier ist die Bedingung `( a pop <= )` also kleiner gleich dem obersten Element von a. Der Stack ist `a popped`, also a ohne das oberste Element. Diese Helferfunktionen sind also für die Spaltung in Teilstacks zuständig.
Daneben gibt es die Hauptfunktion `sort`. Diese ist verzweigt. Wie es für rekursive Funktionen notwendig ist, besitzt sie nämlich eine Abbruchbedingung: Ist die Größe des übergebenen Stacks a (`a size`) kleiner gleich 1, wird der Stack unverändert zurückgegeben. Wenn das nicht der Fall ist, wird der Stack in seine Teilstacks aufgespalten, das heißt `lesser` und `greater` werden darauf angewendet. Für diese Teilstacks wird jeweils nochmals `sort` aufgerufen. So werden die Teilstacks selbst weiter sortiert. Zurückgegeben wird schließlich der sortierte lesser-Stack (`a lesser sort`), erweitert um das erste Element, nach dem die Teilstacks sortiert wurden (`a pop push`) und zusammengefügt mit dem greater-Stack (`a greater sort concat`).

4 Anhang

4.1 Built-In-Funktionen

Mathematisch	Logisch	Datastack
`+` (Addition)	$<$ (kleiner als)	`pop` (das oberste Element eines Datastacks)
`-` (Subtraktion)	$>$ (größer als)	`popped` (ein Datastack ohne das oberste Element)
`*` (Multiplikation)	$=$ (ist gleich)	`is_empty` (ob ein Datastack leer ist)
`/` (Division)	`&` (und)	`push` (legt das zweite Argument auf einen Datastack)
`%` (Modulo)	| (oder)	
	$\sim$ (nicht)	

Literatur

Wikipedia: Funktionale Programmierung. URL: `https://de.wikipedia.org/wiki/Funktionale_Programmierung` (besucht am 27. 02. 2018).