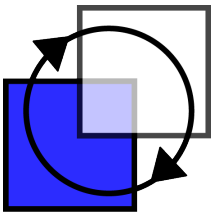
CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

Prof. Bernd Finkbeiner, Ph.D.
Jana Hofmann, M.Sc.
Reactive Systems Group



Programmierung 1 (WS 2020/21)
Zusatzerklärung zum Thema
Faltung

Faltung auf Listen ist eines der wichtigsten Werkzeuge in funktionaler Programmierung, daher wollen wir uns in diesem Dokument ausgiebig damit beschäftigen. Wir starten mit der intuitiven Idee, bevor wir Beispiele anschauen und uns am Ende Übungsaufgaben zur Faltung widmen. Sollten Sie Probleme beim Verständnis des Blattes oder Probleme bei der Bearbeitung der Aufgaben haben, dann besuchen Sie die Office Hour, fragen Sie im Forum oder wenden Sie sich an Ihre(n) Tutor:in.

1 Intuition

Eine erfolgreiche Faltung besteht immer aus drei Komponenten:

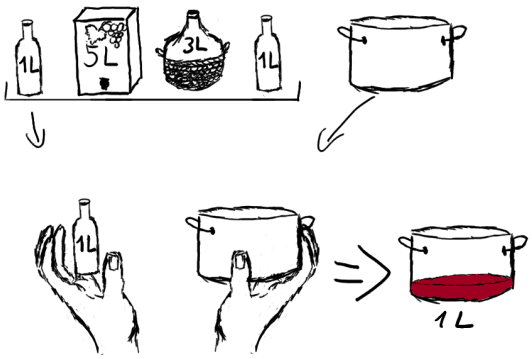
- Eine Abstraktion, welche die Arbeit leistet, die in jedem Schritt zu tun ist.
- Einem Startwert, welcher während des Faltungsvorgangs als Akkumulator fungiert.
- Einer Liste, über die wir falten möchten.

Betrachten wir folgende Situation: Wir haben eine gewisse Anzahl an Glühweinflaschen, in unserem Beispiel genau vier, deren Menge an Inhalt verschieden ist. Zusätzlich einen leeren Topf. Nun möchten wir zum Erhitzen des Glühweins den gesamten Inhalt der Flaschen in den Topf befördern. Diese Aufgabe lässt sich mit Faltung lösen!

Aufgabe 1

Betrachten Sie Abbildung 1. Können Sie – ohne weiterzulesen – bereits die drei wichtigen Komponenten der Faltungsprozedur erkennen?

Abbildung 1



Faltung erledigt genau den Job, den wir intuitiv auch als Mensch tun würden, wenn wir eine Liste von Gegenständen in irgendeiner Form zusammenfassen wollen. Wir nehmen unseren Behälter, den Topf, in die rechte Hand und unsere erste Glühweinflasche in die linke Hand. Dann schütten den Inhalt einfach in den Topf.

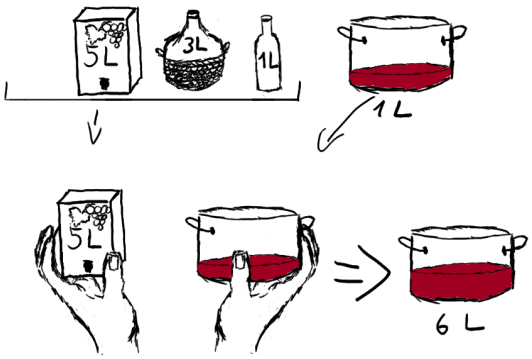
Beachten Sie, dass die Abstraktion genau nach diesem Prinzip aufgebaut ist:

```
1 fn (x, s) ⇒ (* schütte x in s hinein *)
```

Wobei das `x` für das aktuelle Element, in unserem Beispiel für die aktuelle Glühweinflasche in der linken Hand, steht. Das `s` steht für den Akkumulator, in dem Beispiel der Topf. Genau genommen steht der Akkumulator für den Inhalt des Topfes, behalten Sie sich diesen Fakt für später im Hinterkopf.

Haben wir erfolgreich den ersten Schritt der Faltungsprozedur durchlaufen, so ändert sich unser Szenario:

Abbildung 2



Unser Topf-Akkumulator ist nun nicht mehr leer, sondern hat den Wert der ersten Glühweinflasche akkumuliert und enthält nun 1L Glühwein. Außerdem ist die Liste an Glühweinflaschen kleiner geworden und wir machen beim Falten mit der nächsten Flasche weiter. Somit bekommt auch unsere Abstraktion zwei neue Argumente.

Aufgabe 2

Scrollen Sie nicht zur nächsten Seite weiter!

Versuchen Sie die jeweiligen Schritte der Faltung über die Glühweinflaschen nachzuvollziehen, indem Sie die Schritte analog zu den Abbildungen 1 und 2 skizzieren. Identifizieren Sie in jedem Schritt die Argumente der Abstraktion, die Liste mit deren Elementen, und den Akkumulator mit seinen Wert.

In den nächsten Abbildungen sehen Sie, wie wir weiter über die Liste von Glühwein falten.

Abbildung 3

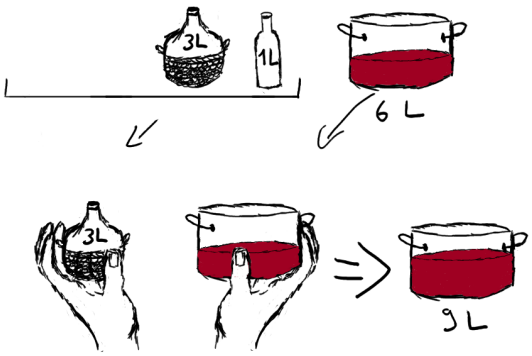
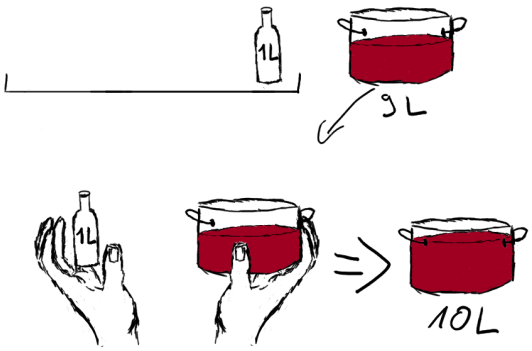


Abbildung 4



Aufgabe 3

Machen Sie sich folgende Sachverhalte klar und übertragen Sie die Intuition nach SML.

- Beachten Sie, dass die Liste nur aus Glühwein bestehen darf; alle Elemente der Liste müssen den gleichen Typ haben. Wir können beispielsweise nicht einfach einen Laib Brot in den Topf werfen!
- Machen Sie sich ebenso klar, dass der Topf nicht unbedingt vom gleichen Typ wie die Elemente der Liste sein muss. Wir können den Glühwein in einen Topf geben, oder wieder in eine Glühweinflasche, aber genauso gut auch in einen Eimer.

  Daher gilt: Vorsicht bei der Wahl des Startwertes, er bestimmt, was am Ende herauskommt!

2 Faltung in SML

Wir haben zwei unterschiedliche Möglichkeiten kennengelernt, um über eine Liste zu falten, nämlich:

- die Faltung von links

```
fun foldl f s nil = s
  | foldl f s (x::xr) = foldl f (f(x, s)) xr
```

- die Faltung von rechts

```
fun foldr f s nil = s
  | foldr f s (x::xr) = f (x, foldr f s xr)
```

`foldl` arbeitet dabei die Liste von links ab, genauso wie wir es oben in dem Glühweinbeispiel gesehen haben. `foldr` arbeitet dagegen die Liste von rechts ab.

Vorsicht: Beachten Sie, dass `foldl` endrekursiv ist, `foldr` jedoch nicht.

Die Funktionsweise der Prozeduren lässt sich sehr gut am Typschema, welches bei beiden Prozeduren gleich ist, ablesen:

$$\forall \alpha, \beta : (\ \alpha\ \texttt{*}\ \beta\ \rightarrow\ \beta\ )\ \rightarrow\ \beta\ \rightarrow\ \alpha\ \texttt{list}\ \rightarrow\ \beta$$

Hinweis: Das Typschema verrät sehr viel über die Funktionsweise einer Prozedur. Schauen Sie sich das Typschema von Prozeduren immer genau an, und prägen Sie es sich ein. Nur so können Sie wissen, wie Sie die Prozeduren mit den richtigen Argumenten aufrufen müssen.

An ihrem Typschema können wir erkennen, dass `foldl` bzw. `foldr` genau die drei wichtigen Komponenten als Argumente erwarten, die wir ganz zu Beginn angesprochen haben:

- Eine Abstraktion als erstes Argument: $(\ \alpha\ \texttt{*}\ \beta\ \rightarrow\ \beta\ )$
  Vergleichen Sie die Abstraktion mit den Händen aus dem Glühweinbeispiel.
- Einen Startwert als zweites Argument: β
- Eine Liste als drittes Argument: $\alpha\ \texttt{list}$

Betrachten Sie außerdem den Rückgabetyp der Prozedur, der ebenfalls vom Typ β ist. Deshalb gilt, wie oben bereits gesagt: Der Startwert, also das zweite Argument, bestimmt, was am Ende aus Ihrer Prozedur herauskommt. Auch die Abstraktion hat den Rückgabetyp β; denken Sie also auch beim Implementieren Ihrer Abstraktion daran!

Aufgabe 4

Wir möchten nun unser Glühweinbeispiel in eine SML-Prozedur übertragen. Wir stellen den Inhalt der Glühweinflaschen durch Werte des Typs `int` dar, ebenso wie den Topf.

- Schreiben Sie mithilfe von `foldl` eine Prozedur `gluehwein : int list → int`, welche die Elemente der Liste aufsummiert.
- Schreiben Sie nun ein Ausführungsprotokoll zu dem Aufruf `gluehwein [1, 5, 3, 1]`.

3 Beispiel

Eine *typische* Aufgabenstellung lautet:

Schreiben Sie mithilfe von `foldl` eine Prozedur `isEven : int list → bool`, die testet, ob alle Elemente einer `int list` gerade sind.

An Aufgaben von diesem Typ können wir immer ähnlich herangehen:

(a) Da wir bereits das Typschema der Prozedur kennen, können wir direkt mit dem Prozedurkopf beginnen:

```
1 fun isEven (xs : int list) : bool =
```

Da die Aufgabenstellung verlangt, `foldl` zu verwenden, können wir den Prozeduraufruf bereits hinschreiben (hier mit Platzhaltern für die drei Argumente von `foldl`):

```
1 fun isEven (xs : int list) : bool = foldl ␣ ␣ ␣
```

(b) Nun betrachten wir den gegebenen Rückgabetyp `bool`. Damit wissen wir schon sehr viel über unsere mögliche Implementierung mit `foldl`. Rufen wir uns nochmals das Typschema von `foldl` in Erinnerung:

$$\forall\alpha,\beta : (\ \alpha\ \texttt{*}\ \beta\ \rightarrow\ \beta\ )\ \rightarrow\ \beta\ \rightarrow\ \alpha\ \texttt{list}\ \rightarrow\ \beta$$

Die Typvariable β muss in unserem Fall mit `bool` instanziiert werden, da das Typschema von `isEven` den Typ `bool` als Rückgabetyp festlegt. Dies bedeutet nun, dass sowohl unser Startwert, als auch der Rückgabetyp der Abstraktion den Typ `bool` haben müssen. Mit `bool` instanziiert würde unser Typschema nun so aussehen:

$$\forall\alpha : (\ \alpha\ \texttt{* bool}\ \rightarrow\ \texttt{bool})\ \rightarrow\ \texttt{bool}\ \rightarrow\ \alpha\ \texttt{list}\ \rightarrow\ \texttt{bool}$$

Nun bleibt nur die Frage zu klären, ob wir als Startwert `true` oder `false` wählen.

Hinweis: Ist man sich bei der Wahl des Startwertes unsicher, ist es unbedingt notwendig, sich zu überlegen, was passieren soll, wenn die Liste von Anfang an leer ist!

In unserem Fall müssten wir uns also überlegen, zu was `isEven []` auswerten soll. In diesem Fall möchten wir `true` zurückgeben, da für alle Elemente in der leeren Liste gilt, dass sie gerade sind. Wenn also keine Elemente in unserer Liste existieren, so ist diese Eigenschaft automatisch erfüllt. Erweitern wir also unsere Implementierung um den Startwert, der an zweiter Stelle hinter `foldl` steht:

```
1 fun isEven (xs : int list) : bool = foldl ␣ true ␣
```

(c) Als nächsten Schritt müssen wir uns überlegen, über welche Liste wir falten möchten. Da wir nur eine einzige Liste als Argument übergeben bekommen haben und dies auch genau die Liste ist, deren Elemente wir durch Faltung betrachten möchten, übergeben wir diese Liste in unserem Aufruf an `foldl`. Unsere Implementierung nimmt also langsam Gestalt an:

```
1 fun isEven (xs : int list) : bool = foldl ␣ true xs
```

(d) Kommen wir nun zum spannendsten Schritt: die Überlegungen rund um die Abstraktion. In der Abstraktion findet immer die Arbeit statt, die in jedem Schritt getan werden muss. Ein Schritt meint dabei die Abarbeitung eines Listenelements. Im Glühweinbeispiel haben wir in die linke Hand die aktuelle Glühweinflasche und in die rechte Hand den Topf genommen und dann den Inhalt der Flasche in den Topf gegeben. Genau so müssen wir nun auch mit unseren Elementen in der Liste und dem Startwert verfahren.

Aus den Überlegungen von oben ergibt sich folgendes Typschema für unsere Abstraktion:

$$\texttt{(int * bool}\ \rightarrow\ \texttt{bool)}$$

Auf der linken Seite des Tupels steht das aktuelle Element der Liste, das wir beim Falten von links nach rechts gerade bearbeiten und auf der rechten Seite des Tupels steht der aktuelle Wert unseres Startwerts. Denken Sie daran, dass der Akku sich – ebenso wie im Glühweinbeispiel – in jedem Schritt ändern kann. Er kann aber auch gleich bleiben. Bezogen auf unser Beispiel wäre das der Fall gewesen, wenn man in einem Schritt eine leere Glühweinflasche in den Topf gegossen hätte.

Wir müssen uns nun überlegen, was wir in jedem Schritt herausfinden möchten. Anfangs ist unser Startwert auf `true` gesetzt und wir müssen uns nun fragen, in welchen Fällen dieser Akkumulator gleich bleiben soll, und wann wir ihn zu `false` ändern wollen. Dabei gibt es für uns zwei Fälle zu betrachten:

- Im ersten Fall haben wir ein Element gefunden, das gerade ist. Dies bedeutet, dass wir unseren Akkumulator auf dem Wert lassen möchten, den er vorher hatte. Wichtig ist dabei, dass es uns egal ist, ob der Startwert vorher auf `true` oder `false` gesetzt war. Wurde vorher beispielsweise schon ein ungerades Element gefunden, dann möchten wir das `false` nicht überschreiben. Wir müssen also bei einem solchen Fall aufpassen:

  ```
  isEven [3,2,4]
  ```

  Am Ende soll dieser Aufruf den Wert `false` liefern, obwohl der letzte Wert, den wir betrachten (nämlich die 4) eine gerade Zahl ist. Die Information, dass irgendwann schon einmal eine ungerade Zahl aufgetreten ist, muss also in jedem Schritt im Akkumulator weitergetragen werden.

- Im zweiten Fall haben wir ein Element gefunden, das ungerade ist. In diesem Fall wissen wir bereits, dass die Liste nicht mehr nur gerade Zahlen enthalten kann und wir möchten somit unseren Startwert auf `false` setzen, dabei kann uns egal sein, welchen Wert er vorher hatte.

Unsere Überlegungen können wir durch folgende Abstraktion realisieren, wobei `a mod 2 = 0` genau die Idee von *a ist gerade* implementiert:

```
1 fn (a, s) ⇒ if a mod 2 = 0 then s
2                            else false
```

Alternativ lässt sich diese Abstraktion auch mit `andalso` realisieren.

```
1 fn (a, s) ⇒ (a mod 2 = 0) andalso s
```

Beachten Sie, dass wir den Startwert `s` einfach weiterreichen, wenn das Element, welches wir gerade betrachten, gerade ist. Wir möchten kein `false` fälschlicherweise überschreiben.

(e) Wir können nun unsere Ideen zu einer Implementierung zusammenfassen, wobei wir nun die expliziten Typangaben weglassen können, da unser SML-Interpreter diese inferieren kann:

```
1 fun isEven xs = foldl (fn (a, s) ⇒ (a mod 2 = 0) andalso s) true xs
```

Aufgabe 5

Fertigen Sie jeweils ein verkürztes Ausführungsprotokoll für folgende Aufrufe von `isEven` an:

(a) `isEven [2, 4, 4]`

(b) `isEven [2, 5, 4]`

Aufgabe 6

Schreiben Sie eine Prozedur `isUneven : int list → bool`, die testet, ob mindestens ein Element der übergebenen Liste ungerade ist.

4 Ein weiteres Beispiel

Eine *typische* Aufgabenstellung lautet:

Schreiben Sie mithilfe von Faltung eine Prozedur `durchschnitt : int list → int`, die den ganzzahligen Durchschnitt aller Zahlen einer Liste berechnet. Verwenden Sie keine zusätzlichen Hilfsprozeduren. Sie dürfen dabei annehmen, dass `durchschnitt nil` zu 0 auswertet.

Diese Aufgabe bedarf ein wenig mehr Überlegungen.

(a) Wir können wieder standardmäßig mit unserer Implementierung beginnen:

```
1 fun durchschnitt xs = foldl ␣ ␣ ␣
```

(b) Für das dritte Argument von `foldl` haben wir in diesem Fall wieder nur eine Wahl, nämlich die Liste `xs`:

```
1 fun durchschnitt xs = foldl ␣ ␣ xs
```

(c) Beim Startwert müssen wir uns in diesem Fall mehr Gedanken machen. Zur Erinnerung: Wir wollen den Durchschnitt der Zahlen in einer Liste berechnen. Dazu müssen wir die Zahlen aufsummieren und sie danach durch die Anzahl der Elemente teilen.

Da in der Aufgabenstellung gefordert ist, dass wir keine zusätzlichen Hilfsprozeduren benutzen dürfen, dürfen wir am Ende auch nicht `List.length` benutzen. Somit müssen wir alle Informationen in einem Durchlauf von `foldl` sammeln. Der wichtige Punkt ist nun, dass es nicht ausreicht, nur ein `int` als Startwert zu wählen, da wir sowohl die Anzahl der Elemente, als auch die Summe der Elemente berechnen müssen. Also behelfen wir uns mit einem Trick: Wir wählen als Startwert ein Tupel. In der ersten Komponente werden wir mitzählen, wie viele Elemente die Liste enthält und in der zweiten Komponente werden wir die Elemente der Liste aufsummieren.

Damit ergibt sich folgender Startwert:

```
1 fun durchschnitt xs = foldl ␣ (0,0) xs
```

(d) Nun kümmern wir uns um die Abstraktion. Wir müssen nun unsere Idee, in der ersten Komponente mitzuzählen, wie viele Elemente die Liste enthält und in der zweiten Komponente die Elemente der Liste aufzusummieren, umsetzen:

```
1 (fn (x, (s, s')) ⇒ (s + 1, s' + x))
```

(e) Wenn wir nun die Implementierung vervollständigen, stoßen wir noch auf ein Problem. Betrachten Sie den Typ unserer Implementierung.

```
1 fun durchschnitt xs = foldl (fn (x, (s, s')) ⇒ (s + 1, s' + x)) (0, 0) xs
```

Momentan hat `durchschnitt` den Typ `int list → (int * int)` und nicht `int list → int`. Außerdem haben wir momentan den Durchschnitt der Elemente der Liste noch nicht berechnet. Um dies zu tun, müssen wir die Summe der Elemente noch durch die Anzahl der Elemente in der Liste teilen. Dies können wir realisieren, indem wir unsere Deklaration in einen `let`-Ausdruck packen und die beiden Komponenten des Ergebnistupels herausprojizieren, um den Durchschnitt zu berechnen.

```
1 fun durchschnitt xs = let
2                           val ergebnisTupel =
3                               foldl (fn (x, (s, s')) ⇒ (s + 1, s' + x)) (0, 0) xs
4                       in
5                           (#2 ergebnisTupel) div (#1 ergebnisTupel)
6                       end
```

5 Übungsaufgaben

Nachdem Sie untenstehende Aufgaben bearbeitet haben, lassen Sie Ihr Ergebnis bei Ihrer/Ihrem Tutor:in oder in der Office Hour überprüfen, oder fragen Sie im Forum nach.

(a) Überlegen Sie sich ähnlich zum Glühweinbeispiel ein Beispiel aus der realen Welt, welches Sie mit Faltung simulieren können. Versuchen Sie sich ähnliche Skizzen zu machen.

(b) Ohne Nachzuschauen! Welches Typschema hat `foldl`? Leiten Sie es sich im Zweifelsfall her.

(c) Schreiben Sie mithilfe von Faltung eine Prozedur `facSum : int list → int`, welche zu jeder Zahl einer Liste von positiven Zahlen die Fakultät berechnet und alle aufsummiert. Beispielsweise soll `facSum [2,4]` zu `2! + 4!`, also 26, auswerten. Schreiben Sie sich eine passende Hilfsprozedur zur Berechnung der Fakultät selbst.

(d) Schreiben Sie die folgenden Prozeduren, ohne zusätzliche Hilfsprozeduren außer den Faltungsprozeduren zu benutzen:

- `onlyTrue : bool list → bool` soll testen, ob in einer Liste nur `true` vorkommt.
- `someTrue : bool list → bool` soll testen, ob in einer Liste mindestens einmal `true` vorkommt.

(e) Schreiben Sie mithilfe von Faltung eine Prozedur `megaPotenz : int list → int`, welche zu einer Liste von Zahlen die Potenz nach folgendem Muster berechnet. Beispielsweise soll `megaPotenz [3,4,7]` zu $3^{(4^7)}$ auswerten.

(f) Schreiben Sie mithilfe von Faltung eine Prozedur `nurGerade : int list → int list`, welche aus einer Liste von Zahlen diejenigen Elemente entfernt, die ungerade sind.

(g) Schreiben Sie mithilfe von Faltung eine Prozedur `hasseEinser : int list → int list`, welche aus einer Liste von Zahlen diejenigen Elemente entfernt, welche die Ziffer 1 enthalten. Beispielsweise soll `hasseEinser [1, 22, 12, 11, 353, 20001]` zu `[22, 353]` auswerten.

(h) Schreiben Sie mithilfe von Faltung eine Prozedur `miniSum : int → int list → int list`. Die Prozedur soll gegeben einer Zahl `n` eine Liste von Zahlen so zu einer neuen Liste umwandeln, dass jeweils die `n` aufeinander folgenden Zahlen aufsummiert wurden. So soll beispielsweise `miniSum 3 [2, 5, 7, 9, 9, 9]` zu `[14, 27]` auswerten.

(i) Denken Sie sich selbst beliebige Aufgaben aus, die sich mit Faltung lösen lassen. Sie sollten die Aufgabe selbst bearbeiten können, oder sogar eine Beispiellösung erstellen können. Posten Sie Ihre selbst erstellten Aufgaben im Forum und lassen Sie Ihre Kommilitonen knobeln.