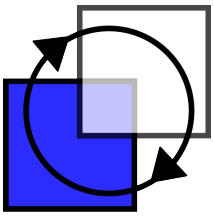
CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

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



Programmierung 1 (WS 2020/21)
Zusatzerklärung zum Thema
iter – Die bestimmte Iteration

1 Die Funktionsweise

Was tut `iter`? - `iter` ist eine endrekursive Hilfsprozedur, die es ermöglicht, viele rekursive Probleme zu lösen. `iter` wendet dafür eine übergebene Prozedur `f` genau `n`-mal auf einen Startwert an.

Zur Erinnerung hier noch einmal die Deklaration von iter:

```
1 fun iter n s f = if n < 1 then s else iter (n − 1) (f s) f
```

Um zu verstehen, was das bedeutet betrachte man zunächst die Argumente, die `iter` übergeben werden:

1.1 Die Argumente

Man rufe sich das Typschema von `iter` in Erinnerung:

`iter`: $\forall\alpha$. `int` $\rightarrow$ α $\rightarrow$ (α $\rightarrow$ α) $\rightarrow$ α

`iter` übernimmt nach diesem Schema kaskadiert drei Argumente:

(a) Eine Ganzzahl `n` vom Typ `int`, die beschreibt, wie häufig das zu Tuende ausgeführt werden soll. (Was genau das ist, folgt!).

(b) Ein Startwert mit dem polymorphen Typ α. Dieser Startwert wird im Fall, dass als erstes Argument eine 0 übergeben wurde - die Ausführung folglich 0-mal durchgeführt werden soll - direkt zurück gegeben. Ansonsten wird dieser Startwert in jedem Schritt mit Hilfe des letzten Argumentes modifiziert.

(c) Das letzte Argument ist ein sehr wichtiger Bestandteil der Prozedur `iter`. Es handelt sich um eine Prozedur `f`: $\alpha \rightarrow \alpha$.

Die Tatsache, dass eine Prozedur selbst als Argument übergeben wird, macht `iter` höherstufig. Diese übergebene Prozedur beschreibt, wie der Startwert in jedem Schritt verändert wird. Man rufe sich das Typschema in Erinnerung: Die Prozedur muss ein Argument vom gleichen Typ des Startwerts nehmen und einen Wert des gleichen Typs zurückliefern. Dies ermöglicht eine mehrfache Anwendung von `f`, beispielsweise liefert eine Prozeduranwendung mit `n = 3` für eine Prozedur `f` und einen Startwert `s` den folgenden Wert: `f(f(f(s)))`. `f` wird im ersten Schritt auf `s` angewendet. Auf das daraufhin erhaltene Ergebnis wird `f` wiederum angewendet... Deshalb müssen Argument- und Ergbnistyp von `f` identisch sein.

1.2 Prozedurrumpf

Die Umsetzung des beschriebenen Vorgehens (also `n`-fache Anwendung von `f` auf `s`) ist im Prozedurrumpf wie folgt umgesetzt.

Zunächst wird überprüft, ob `n` bereits den Wert 0 erreicht hat (`if n < 1`). Ist dies der Fall, so soll eine 0-fache Anwendung von `f` auf `s` stattfinden. Wenn `f` nie auf `s` angewendet wird, so wird `s` auch nicht verändert. `s` kann folglich direkt unverändert zurückgegeben werden. Spannender wird der Fall, in dem `n` größer 0 ist. In diesem Fall wird in Rekursion gegangen. Man wendet `f` einmal auf `s` an (`f s`), und übergibt diesen Wert an `iter` als neuen Startwert. Zusätzlich verringert man `n` um 1.

Intuitiv bedeutet dies:

`f` genau `n`-mal auf `s` anzuwenden, ist das gleiche, wie `f` genau `n−1`-mal auf `f s` anzuwenden, da `f s` ja bereits der einfachen Anwendung entspricht.

Auf diese Weise kann das Problem Schritt für Schritt verkleinert werden, bis man beim einfachsten Fall, der 0-fachen Anwendung von `f` auf `s` angekommen ist.

1.3 Ein einfaches Beispiel

Das eben Beschriebene klingt nun noch recht abstrakt, daher betrachten wir nun ein Beispiel.

Es seien gegeben: $n \coloneqq 2, s \coloneqq 3, f \coloneqq$ (`fun f x = x + 1, int → int`, []).

Nach der obigen Erklärung soll `f` insgesamt zweimal(n) auf 3(s) angewendet werden. Die bedeutet hier konkret:

```
f(f(3)) = f(3 + 1) = f(4) =  4 + 1 = 5
```

Da `f` eine Prozedur ist, die 1 auf eine gegebene Zahl addiert, bedeutet die zweifache Anwendung von `f` auf 3, dass zunächst 1 auf 3 addiert wird, und auf das daraus resultierende Ergebnis (hier 4) noch einmal 1 addiert wird. Effektiv wird `(3 + 1) + 1` berechnet.

2 Anwendung

Viele rekursive Probleme lassen sich durch die Verwendung von `iter` lösen. Im Folgenden wird gezeigt, wie man an Aufgaben herangehen kann, die verlangen, neue Prozeduren mithilfe von `iter` zu schreiben:

2.1 iter pur

Bei dem Beispiel aus dem vorherigen Abschnitt haben wir die Argumente von `iter` isoliert betrachtet. Im Allgemeinen wird `iter` benutzt um neue Prozeduren zu schreiben.

Ein Beispiel dafür ist die Multiplikation zweier Ganzzahlen:

```
mul : int → int → int
```

Unser Ziel ist es, diese Funktionalität zu implementieren, indem wir `iter` mit geeigneten Argumenten aufrufen.

Diese Argumente können dabei auch aus den Argumentbezeichnern der gerade zu schreibenden Prozedur gewählt werden:

Ist der Beginn einer Prozedurdeklaration für `mul` wie folgt gegeben:

```
fun mul x y = ...
```

so dürften für den Aufruf von `iter` auch `x` und `y` verwendet werden. Es ist hilfreich, sich zunächst informell zu überlegen, wie sich zwei Zahlen iterativ (sprich: schrittweise, hier mithilfe von Addition) miteinander multiplizieren lassen: $2 \cdot 3$ lässt sich beispielsweise auch als $3 + 3$ schreiben. $4 \cdot 3$ als $3 + 3 + 3 + 3$. $x \cdot y$ ergibt sich folglich durch x-malige Addition von y. Ausgehend von dieser Überlegung lassen sich die Rekursionsgleichungen aufstellen:

```
0 * y = 0
x * y = y + (x − 1) * y
```

Anders als beim Bauen einer eigenen rekursiven Prozedur leitet man aus diesen Gleichungen die Werte für `n`, `s` und `f` ab.

- Das `n` zu bestimmen kann knifflig werden. Denn nicht jede rekursive Prozedur lässt sich mit `iter` darstellen, sondern nur diejenigen, von denen vorher bekannt ist, wie häufig der Rekursionsschritt ausgeführt werden soll.

  Durch die intuitive Beschreibung lässt sich hier jedoch gut erschließen, dass wir den Wert von `y` genau `x`-mal addieren wollen. Die Addition soll folglich `x`-mal ausgeführt werden. Wir wählen unser `n` als `x`.

- Unseren Startwert erhalten wir aus der Gleichung für den Basisfall:

  Die 0-malige Anwendung soll laut der aufgestellten Gleichung das Ergebnis 0 liefern. Der Startwert muss dementsprechend 0 sein, da er dem entspricht, was für $n = 0$ (bzw. hier $x = 0$) zurückgeliefert wird.

- Die Prozedur `f` ergibt sich aus der Gleichung für den rekursiven Aufruf. Diese beschreibt, wie man aus dem Ergebnis der `(n − 1)`-maligen Anwendung von `f`, auf das Ergebnis der `n`-maligen Anwendung kommt. (In diesem Fall gelingt dies durch die Addition von `y`.)

Man baut nun eine Abstraktion, die einen Wert vom Typ `int` nimmt und einen Wert vom Typ `int` zurückliefert. Das Argument kann dabei als Akku betrachtet werden, der bereits den Wert für $(x-1)\cdot y$ enthält. Die Abstraktion hat es nun zum Ziel aus $(x-1)\cdot y$ das Produkt $x\cdot y$ zu berechnen. Laut Rekursionsgleichung gelingt dies durch die Addition von `y`.

Die Abstraktion muss insgesamt wie folgt lauten:

```
1 fn (a : int) ⇒ a + y
```

Die vollständige Prozedurdeklaration ergibt sich damit wie folgt:

```
1 fun mul x y = iter x 0 (fn a ⇒ a + y)
```

Was passiert nun bei dem Aufruf `mul 2 3`?

Der Aufruf von `iter` wird in der Umgebung $[x \coloneqq 2, y \coloneqq 3]$ ausgeführt.

Wir betrachten das folgende verkürzte Ausführungsprotokoll:

```
1 mul 2 3 = iter 2 0 (fn a ⇒ a + 3)
2         = iter 1 ((fn a ⇒ a + 3) 0) (fn a ⇒ a + 3)
3         = iter 1 (0 + 3) (fn a ⇒ a + 3)
4         = iter 1 3 (fn a ⇒ a + 3)
5         = iter 0 ((fn a ⇒ a +3) 3) (fn a ⇒ a + 3)
6         = iter 0 (3 + 3) (fn a ⇒ a + 3)
7         = iter 0 6 (fn a ⇒ a + 3)
8         = 6.
```

Es wird 2 mal 3 auf den Startwert 0 addiert, was genau der 2-maligen Anwendung der Abstraktion `(fn a ⇒ a + 3)` auf den Startwert 0 entspricht.

2.2 iter mit Tupeln

Es kann passieren, dass sich das obige Schema nicht direkt anwenden lässt. Wir betrachten das Beispiel, bei dem die n-te Fakultät mit Hilfe von `iter` berechnet werden soll.

```
1 fun fac n = ...
```

Für ein gegebenes n soll folglich das folgende Produkt berechnet werden: $1\cdot 2\cdot \ldots \cdot n$.

In jedem Schritt muss - bei 1 beginnend - eine um 1 erhöhte Zahl mit dem bisherigen Ergebnis multipliziert werden.

Die Rekursionsgleichungen lauten:

```
1 fac 0 = 1
2 fac n = n * fac(n − 1)
```

Es werden n Multiplikationsschritte benötigt, die Schrittanzahl ist folglich durch `n` gegeben. Der Startwert ergibt sich direkt aus der ersten Rekursionsgleichung und ist 1.

Wie sieht nun die Abstraktion aus? Verfährt man wie bei dem Beispiel der Multiplikation gesehen, so beginnen wir mit `fn a ⇒ ?`, wobei `a` als Akkumulator betrachtet werden kann, der bereits das Ergebnis von `fac(n − 1)` beinhaltet.

Die Berechnung von `fac n` müsste laut Rekursionsvorschrift durch die Multiplikation von `n` erfolgen. Dementsprechend könnte man auf die Idee kommen, dass die Abstraktion wie folgt lauten müsste:

```
1 fn a ⇒ n * a
```

Achtung: An dieser Stelle ist es wichtig sich genau klar zu machen, an welchen Wert `n` gebunden ist. `n` ist das von der Prozedur `fac` übergebene Argument. Beim Aufruf `fac 3` wird `n` als im Folgenden immer an 3 gebunden. Die Abstraktion, mit der `iter` aufgerufen wird, lautet damit effektiv wie folgt:

```
1 fn a ⇒ 3 * a
```

Das verkürzten Ausführungsprotokoll gibt nun Aufschluss über die tatsächliche Funktionalität unserer Prozedur `fac`.

```
iter 3 1 (fn a ⇒ 3 * a) = iter 3  1 (fn a ⇒ 3 * a)
                        = iter 2  3 (fn a ⇒ 3 * a)
                        = iter 1  9 (fn a ⇒ 3 * a)
                        = iter 0 27 (fn a ⇒ 3 * a)
                        = 27
```

Die Prozedur berechnet also 3^3 und nicht 3!.

Woran liegt das? Bei jedem Schritt wird das Ergebnis mit 3 multipliziert. Das `n`, welches jedoch tatsächlich multipliziert werden sollte, müsste variieren. Es sollte in der ersten Iteration den Wert 1, in der zweiten Iteration den Wert 2 usw. haben...

Die einzige Information, die in der Abstraktion zur Verfügung steht, ist der Wert von `a`, welcher das Ergebnis der um eins verringerter Größe repräsentiert. Welche Größe das jedoch genau ist, lässt sich daraus nicht ablesen.

Es fehlt also eine essentielle Information, um zu beschreiben, wie man vom Ergebnis der $(n-1)$-maligen Anwendung zum Ergebnis der n-maligen Anwendung kommt - nämlich wie das n im konkreten Fall aussieht.

Die beste Möglichkeit, dieses Problem zu lösen, ist ein „Speichern“ der fehlenden Information, bzw. diese in jedem Schritt mitzutragen. Wie kann dies umgesetzt werden, wenn `iter` nur *einen* Startwert übergeben bekommt, der in jedem Schritt modifiziert werden darf?

An dieser Stelle hilft die polymorphe Typisierung von `iter`. Diese erlaubt es, für das α unseres polymorphen Typschemas auch einen Tupeltypen einzusetzen. Somit können wir ein Paar aus zwei Zahlen als Startwert wählen, wobei eine Komponente einen Zähler, die andere - wie gewohnt - unseren Akku speichert. Welche Auswirkungen hat das auf die Wahl von `n` und `s`?

`n` ist monomorph auf `int` getypt. Hier ändert sich folglich nichts. Der Startwert hingen muss angepasst werden.

In der ersten Komponente soll der Zähler gespeichert werden, der mit jeder Iteration erhöht wird. Dieser soll gewährleisten, dass bei der ersten Iteration der Startwert mit 1, in der zweiten mit 2 etc. multipliziert werden kann. Dementsprechend muss dieser Wert zu Beginn mit 1 belegt werden. Die zweite Komponente muss nach der obigen Überlegung ebenfalls mit 1 initialisiert werden. Es ergibt sich der Startwert $(1, 1)$.

Wie kann die Prozedur angepasst werden?

Die Abstraktion muss nach dem Typschema von `iter` etwas vom gleichen Typ wie dem Startwert – ein Paar aus `int*int` – als Argument nehmen und auch ein solches wieder zurück geben. Die erste Komponente des übergebenen Paars enthält die Information darüber, in der wievielten Iteration man sich befindet. Da bei 1 begonnen wird, muss dieser Wert bei jeder Iteration hoch gezählt werden. Die zweite Komponente enthält den Akku, das Ergebnis für die bisher durchgeführten Iterationen. Der neue Wert entspricht der Multiplikation der zweiten Komponente mit dem in der ersten Komponente übergebenen Zähler.

Damit muss die Abstraktion wie folgt lauten:

```
fn (n', a) ⇒ (n' + 1, a * n')
```

Wir sind nun beinahe fertig. Es gilt noch zu beachten, von welchem Typ das Ergebnis ist, das `iter` liefert. Laut dem Typschema von `iter` ist dieses vom gleichen Typ wie der Startwert – in diesem Fall ein Paar.

Die Prozedur `fac` soll jedoch nur einen Wert vom Typ `int` zurück liefern. Nun gilt es, sich zu überlegen, welche Komponente wir zurückgeben wollen.

In der ersten Komponente befindet sich lediglich der Zähler. Er sollte nach `n`-maliger Iteration den Wert `n` haben. In der zweiten Komponente liegt der Akku, der das gewünschte Ergebnis speichert. Diesen soll die Prozedur zurück geben.

Mithilfe einer Projektion kann darauf zugegriffen werden. Damit ist die gesamte Prozedur wie folgt gegeben:

```
fun fac n = #2 ( iter n (1, 1) (fn(n', a) ⇒ (n' + 1, a * n')) )
```