Hintergrund und Grundlagen

Aufgreifen von...

- Auswertung von Ausdrücken

  Schlagwörter: applicative und normal order evaluation, eager und lazy evaluation, ...

  ...siehe Newsgruppenbeitrag vom 15.10.2003

- λ-Kalkül

  ...formale Fundierung nahezu aller funktionalen Programmiersprachen

Zur Erinnerung...

$$\text{fact } n = \text{if } n == 1 \text{ then } 1 \text{ else } (n * \text{fact } (n - 1))$$

$$\begin{aligned}
\text{fact } 3 &\Rightarrow \text{if } 3 == 1 \text{ then } 1 \text{ else } (3 * \text{fact } (3 - 1)) \\
&\Rightarrow 3 * \text{fact } (3 - 1) \\
&\Rightarrow 3 * \text{fact } 2 \\
&\Rightarrow 3 * (\text{if } 2 == 1 \text{ then } 1 \text{ else } (2 * \text{fact } (2 - 1))) \\
&\Rightarrow 3 * (2 * \text{fact } (2 - 1)) \\
&\Rightarrow 3 * (2 * \text{fact } 1) \\
&\Rightarrow 3 * (2 * (\text{if } 1 == 1 \text{ then } 1 \text{ else } (1 * \text{fact } (1 - 1)))) \\
&\Rightarrow 3 * (2 * 1) \\
&\Rightarrow 3 * 2 \\
&\Rightarrow 6
\end{aligned}$$

Im Detail...

$\text{fact } n = \text{if } n == 1 \text{ then } 1 \text{ else } (n * \text{fact } (n - 1))$

$\text{fact } 3 \Rightarrow \text{if } 3 == 1 \text{ then } 1 \text{ else } (3 * \text{fact } (3 - 1))$

$\Rightarrow 3 * \text{fact } (3 - 1)$

Weiter mit a)

$\Rightarrow 3 * \text{fact } 2$

$\Rightarrow 3 * (\text{if } 2 == 1 \text{ then } 1 \text{ else } (2 * \text{fact } (2-1)))$

$\Rightarrow$ wie bisher...

...oder mit b)

$\Rightarrow 3 * (\text{if } (3-1) == 1 \text{ then } 1 \text{ else } ((3-1) * \text{fact } ((3-1)-1)))$

$\Rightarrow$ analog fortgeführt...

Schließlich...

$\Rightarrow 6$

Freiheitsgrade...

Betrachte...

```
fact(fact(square(2+2))) * fact(fact(square(3)))
```

Zentral...

- **Wo** im Ausdruck mit der Auswertung fortfahren?
- **Wie** mit (Funktions-) Argumenten umgehen?

Gretchenfrage...

- **Welcher** Einfluss auf das Ergebnis?

Glücklicherweise...

Theorem

Jede terminierende Auswertungsreihenfolge endet mit demselben Ergebnis.

...Alonzo Church/J. Barclay Rosser (1936)

Auswertungsstrategien...

In der Praxis...

Um den Ausdruck $f(exp)$ auszuwerten...

a) ...berechne zunächst den Wert von exp (und setze diesen Wert dann im Rumpf von f ein)
   - $\leadsto$ applicative order evaluation, eager evaluation, call-by-value evaluation, leftmost-innermost evaluation

b) ...setze exp unmittelbar im Rumpf von f ein und werte den so entstehenden Ausdruck aus
   - $\leadsto$ normal order evaluation, call-by-name evaluation, leftmost-outermost evaluation
   - $\leadsto$ *Verfeinerung*: lazy evaluation, call-by-need evaluation

Ein Beispiel...

Einige einfache Funktionen...

```
-- Die Funktion square zur Quadrierung einer ganzen Zahl
square :: Int -> Int
square n = n*n

-- Die Funktion first zur Projektion auf die erste Paarkomponente
first :: (Int,Int) -> Int
first (m,n) = m

-- Die Funktion infiniteInc zum "ewigen" Inkrement
infiniteInc :: Int
infiniteInc = 1 + infiniteInc
```

Auswertung gemäß...

...applicative order (leftmost-innermost):

$$
\begin{aligned}
& \texttt{square}(\texttt{square}(\texttt{square}(2))) \\
\Rightarrow\ & \texttt{square}(\texttt{square}(2 * 2)) \\
\Rightarrow\ & \texttt{square}(\texttt{square}(4)) \\
\Rightarrow\ & \texttt{square}(4 * 4) \\
\Rightarrow\ & \texttt{square}(16) \\
\Rightarrow\ & 16 * 16 \\
\Rightarrow\ & 256
\end{aligned}
$$

...6 Schritte.

Auswertung gemäß...

...normal order (leftmost-outermost):

$$
\begin{aligned}
& \texttt{square(square(square(2)))} \\
\Rightarrow\ & \texttt{square(square(2))} * \texttt{square(square(2))} \\
\Rightarrow\ & (\texttt{square(2)} * \texttt{square(2))} * \texttt{square(square(2))} \\
\Rightarrow\ & ((2 * 2) * \texttt{square(2)}) * \texttt{square(square(2))} \\
\Rightarrow\ & (4 * \texttt{square(2)}) * \texttt{square(square(2))} \\
\Rightarrow\ & (4 * (2 * 2)) * \texttt{square(square(2))} \\
\Rightarrow\ & (4 * 4) * \texttt{square(square(2))} \\
\Rightarrow\ & 16 * \texttt{square(square(2))} \\
\Rightarrow\ & \ldots \\
\Rightarrow\ & 16 * 16 \\
\Rightarrow\ & 256
\end{aligned}
$$

...1+6+6+1=14 Schritte.

Applicative order effizienter?

Nicht immer...

$$\texttt{first}\ (42, \texttt{square}(\texttt{square}(\texttt{square}(2))))$$

...in applicative order

$$\begin{aligned} & \texttt{first}\ (42, \texttt{square}(\texttt{square}(\texttt{square}(2)))) \\ \Rightarrow\ & \ldots \\ \Rightarrow\ & \texttt{first}\ (42, 256) \\ \Rightarrow\ & 42 \end{aligned}$$

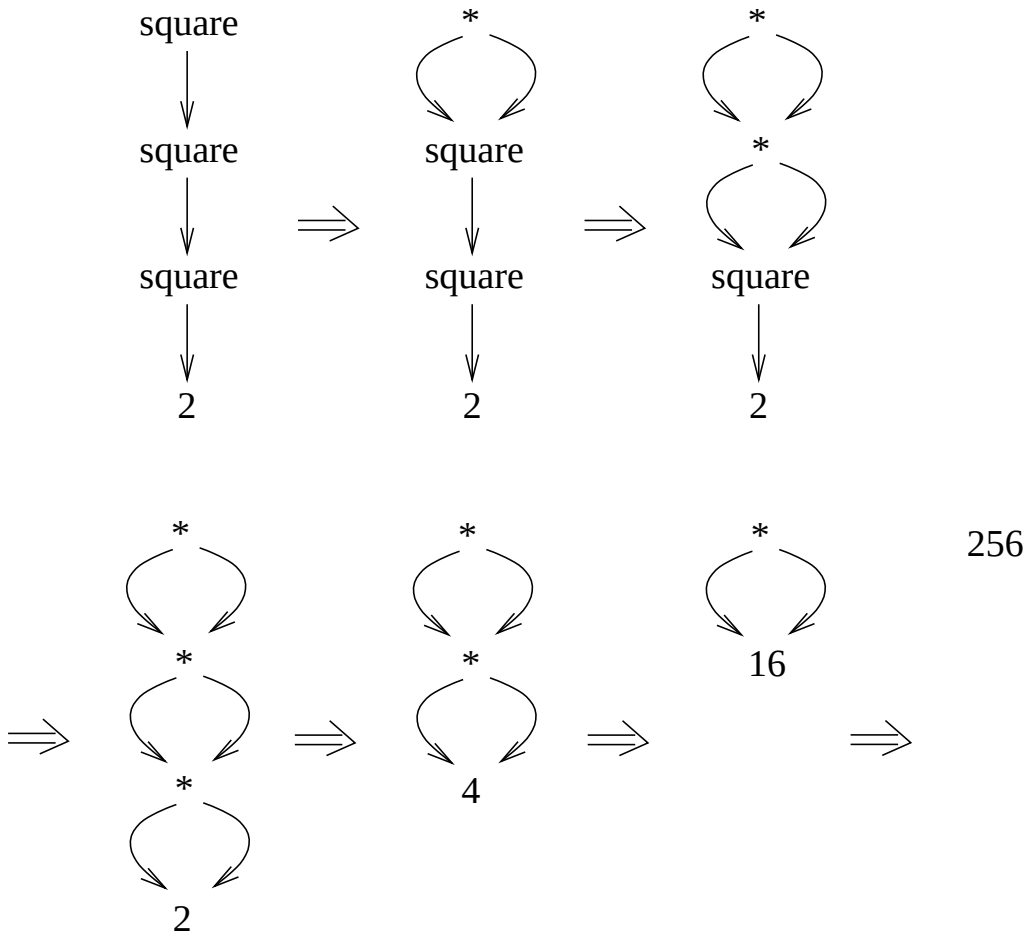
...1+6+1=8 Schritte.

...in normal order

$$\begin{aligned} & \texttt{first}\ (42, \texttt{square}(\texttt{square}(\texttt{square}(2)))) \\ \Rightarrow\ & 42 \end{aligned}$$

...1 Schritt.

Lazy evaluation (call-by-need)...



...6 Schritte, aber Graphtransformationen!

Lazy evaluation (call-by-need)...

- ...beruht auf Graphtransformationen
- ...garantiert, dass Argumente höchstens einmal (möglicherweise also gar nicht) ausgewertet werden

Insgesamt...

$\leadsto$ ...effiziente Realisierung der normal order Strategie!

Zentrale Ergebnisse...

Theorem

- ...alle terminierenden Auswertungsreihenfolgen enden mit demselben Ergebnis

  $\leadsto$ Konfluenz- oder Diamanteigenschaft

- ...wenn es eine terminierende Auswertungsreihenfolge gibt, so terminiert auch die normal order Auswertungsreihenfolge

...Church/Rosser (1936)

Insbesondere:
Lazy evaluation "vergleichbar effizient" wie applicative (eager) order, falls alle Argumente benötigt werden.

Frei nach Shakespeare...

Eager or lazy evaluation: that is the question.

Quot capita, tot sensa.

Oder: Die Antworten sind verschieden...

- eager evaluation (z.B. ML, Scheme (abgesehen von Makros),...)
- lazy evaluation (z.B. Haskell, Miranda,...)

Lazy vs. Eager: Eine Abwägung... (1)

Lazy Evaluation

- Vorteile
  - terminiert mit Normalform, wenn es eine terminierende Auswertungsreihenfolge gibt
  - wertet Argumente nur aus, wenn nötig
  - elegante Behandlung potentiell unendlicher Datenstrukturen
- Nachteile
  - konzeptuell und implementierungstechnisch anspruchsvoller
    - partielle Auswertung von Ausdrücken (Seiteneffekte! Beachte: Letztere nicht in Haskell! In Scheme: Verantwortung beim Programmierer.)
    - Graphtransformationen
    - Ein-/Ausgabe
  - Volles Verständnis: Domain-Theorie und λ-Kalkül im Detail

Lazy vs. Eager: Eine Abwägung... (2)

Eager Evaluation

- Vorteile
  - Konzeptuell und implementierungstechnisch einfacher
  - Vom mathematischen Standpunkt oft "natürlicher" (Beispiel: `first (42,infiniteInc)`)
  - (Einfache(re) Integration imperativer Konzepte)

Mithin... eager oder lazy – eine Frage des Anwendungsprofils!

Zu guter Letzt...

Ist ein Haskell-Compiler (Interpretierer) korrekt, der die Fakultätsfunktion applikativ auswertet?

Ja, weil die Funktion `fact` *strikt* in ihrem Argument ist...

$\leadsto$ eager evaluation oder auch strict evaluation!