$$
\begin{array}{l}
\text{for } (i = 0; i < N; i{+}{+}) \\
\quad \text{for } (j = 0; j < M; j{+}{+}) \; \{ \\
\quad\quad c[i][j] = 0; \\
\quad\quad \text{for } (k = 0; k < K; k{+}{+}) \\
\quad\quad\quad c[i][j] = c[i][j] + a[i][k] \cdot b[k][j]; \\
\quad \}
\end{array}
$$

- Jetzt können wir die beiden Iterationen nicht einfach vertauschen :-(
- Wir können aber die Iteration über j duplizieren ...

$$\text{for } (i = 0; i < N; i{+}{+}) \; \{$$
$$\quad \text{for } (j = 0; j < M; j{+}{+}) \; c[i][j] = 0;$$
$$\quad \text{for } (j = 0; j < M; j{+}{+})$$
$$\quad\quad \text{for } (k = 0; k < K; k{+}{+})$$
$$\quad\quad\quad c[i][j] = c[i][j] + a[i][k] \cdot b[k][j];$$
$$\}$$

Zur Korrektheit:

$\Longrightarrow$ Die gelesenen Einträge (hier: keine) dürfen im Rest des Rumpfs nicht modifiziert werden !!!

$\Longrightarrow$ Die Reihenfolge der Schreibzugriffe einer Zelle darf sich nicht ändern :-)

Man erhält:

$$
\begin{array}{l}
\textsf{for}\ \ (i = 0; i < N; i{+}{+})\ \ \{ \\
\qquad \textsf{for}\ \ (j = 0; j < M; j{+}{+})\ \ c[i][j] = 0; \\
\qquad \textsf{for}\ \ (k = 0; k < K; k{+}{+}) \\
\qquad\qquad \textsf{for}\ \ (j = 0; j < M; j{+}{+}) \\
\qquad\qquad\qquad c[i][j] = c[i][j] + a[i][k] \cdot b[k][j]; \\
\}
\end{array}
$$

Diskussion:

- Statt mehrere Schleifen zusammen zu fassen, haben wir Schleifen distribuiert :-)
- Desgleichen zieht man Abfragen vor die Schleife $\Longrightarrow$ if-Distribution ...

Achtung:

Statt dieser Transformation könnte man die innere Schleife auch anders optimieren:

$$
\begin{array}{l}
\textsf{for}\ \ (i = 0; i < N; i{+}{+}) \\
\qquad \textsf{for}\ \ (j = 0; j < M; j{+}{+})\ \ \{ \\
\qquad\qquad t = 0; \\
\qquad\qquad \textsf{for}\ \ (k = 0; k < K; k{+}{+}) \\
\qquad\qquad\qquad t = t + a[i][k] \cdot b[k][j]; \\
\qquad\qquad c[i][j] = t; \\
\qquad \}
\end{array}
$$

Idee:

Finden wir ein heftig benutztes Feld-Element $a[e_1] \ldots [e_r]$, dessen Index-Ausdrücke e_l innerhalb der inneren Schleife konstant sind, können wir stattdessen ein Hilfsregister spendieren :-)

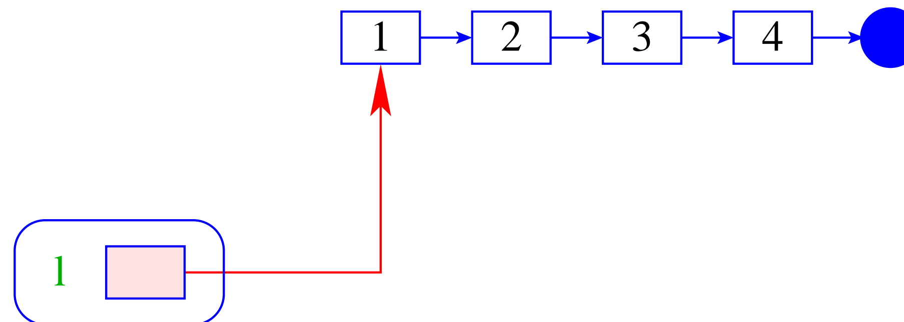
Achtung:

Diese Optimierung verhindert die vorherige und umgekehrt ...

Diskussion:

- Die bisherigen Optimierungen beziehen sich auf Iterationen über Feldern.
- Cache-sensible Organisation anderer Datenstrukturen ist möglich, aber i.a. nicht vollautomatisch möglich ...

Beispiel: Keller



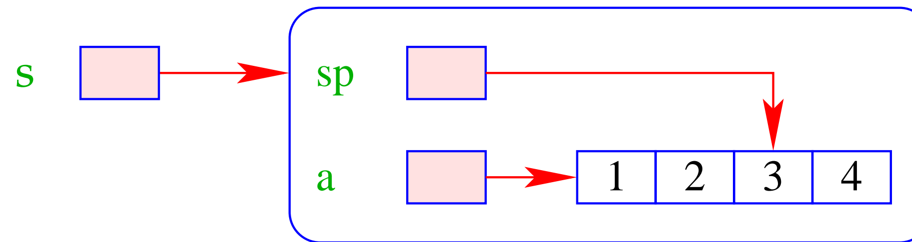
Vorteil:

- \+ Die Implementierung ist einfach :-)
- \+ Die Operationen push / pop erfordern konstante Zeit :-)
- \+ Die Datenstruktur ist potentiell beliebig groß :-)

Nachteil:

- – Die einzelnen Listenknoten können beliebig über den Speicher verteilt sein :-(

Alternative:



Vorteil:

+ Die Implementierung ist auch einfach :-)
+ Die Operationen push / pop erfordern konstante Zeit :-)
+ Die Daten liegen konsequtiv; Stack-Schwankungen sind im Mittel gering

$\Longrightarrow$ gutes Cache-Verhalten !!!

Nachteil:

– Die Datenstruktur ist beschränkt :-(

Verbesserung:

- Ist das Feld voll, ersetze es durch ein doppelt so großes !!!
- Wird das Feld leer bis auf ein Viertel, halbiere es wieder !!!

$\Longrightarrow$ Die Extra-Kosten sind amortisiert konstant :-)

$\Longrightarrow$ Die Implementierung ist nicht mehr ganz so trivial :-}

Diskussion:

- → Die gleiche Idee klappt auch für Schlangen :-)
- → Andere Datenstrukturen bemüht man sich, blockweise aufzuteilen.

  Problem: wie organisiert man die Zugriffe, dass sie möglichst lange auf dem selben Block arbeiten ???

$\Longrightarrow$ Algorithmen auf externen Daten

2. Stack-Allokation statt Heap-Allokation

Problem:

- Programmiersprachen wie Java legen alle Datenstrukturen im Heap an — selbst wenn sie nur innerhalb der aktuellen Methode benötigt werden :-(
- Überlebt kein Verweis auf diese Daten den Aufruf, wollen wir sie auf dem Stack allokieren :-)

$\Longrightarrow$ Escape-Analyse

Idee:

Berechne Alias-Information.

Bestimme, ob ein erzeugtes Objekt möglicherweise von außen erreichbar ist ...

Beispiel: unsere Pointer-Sprache

$$x = \mathsf{new}();$$
$$y = \mathsf{new}();$$
$$x \to a = y;$$
$$z = y;$$
$$\mathsf{return}\ \ z;$$

... könnte ein möglicher Methoden-Rumpf sein ;-)

Von außen zugänglich sind Objekte, die:

- von `return` zurück geliefert werden;
- einer globalen Variablen zugewiesen werden;
- von solchen Objekten erreichbar sind.

... im Beispiel:

$$
\begin{aligned}
& x = \mathsf{new}(); \\
& y = \mathsf{new}(); \\
& x \to a = y; \\
& z = y; \\
& \mathsf{return}\ \boxed{z};
\end{aligned}
$$

Von außen zugänglich sind Objekte, die:

- von return zurück geliefert werden;
- einer globalen Variablen zugewiesen werden;
- von solchen Objekten erreichbar sind.

... im Beispiel:

$$x = \mathsf{new}();$$
$$y = \mathsf{new}();$$
$$x \rightarrow a = y;$$
$$z = \boxed{y};$$
$$\mathsf{return}\ \boxed{z};$$

Von außen zugänglich sind Objekte, die:

- von return zurück geliefert werden;
- einer globalen Variablen zugewiesen werden;
- von solchen Objekten erreichbar sind.

... im Beispiel:

$$x = \mathsf{new}();$$
$$y = \mathsf{new}();$$
$$\boxed{x \longrightarrow a} = y;$$
$$z = \boxed{y};$$
$$\mathsf{return}\ \ \boxed{z};$$

Von außen zugänglich sind Objekte, die:

- von return zurück geliefert werden;
- einer globalen Variablen zugewiesen werden;
- von solchen Objekten erreichbar sind.

... im Beispiel:

$$
\begin{aligned}
& x = \mathsf{new}(); \\
& y = \boxed{\mathsf{new}()}; \\
& \boxed{x \longrightarrow a} = y; \\
& z = \boxed{y}; \\
& \mathsf{return}\ \boxed{z};
\end{aligned}
$$

Wir schließen:

- Die Objekte, die das erste `new()` anlegt, können nicht entkommen.
- Wir können sie darum auf dem Stack allokieren :-)

Achtung:

Das ist natürlich nur sinnvoll, wenn von dieser Sorte nur wenige pro Methoden-Aufruf angelegt werden :-(

Liegt deshalb ein solches lokales `new()` in einer Schleife, sollten wir die Objekte vorsichtshalber doch im Heap anlegen ;-)

Erweiterung: Prozeduren

- Wir benötigen eine interprozedurale Alias-Analyse :-)
- Kennen wir das gesamte Programm, können wir z.B. die Kontrollflussgraphen der einzelnen Prozeduren zu einem einzigen zusammen fassen (durch Hinzufügen geeigneter Kanten) und für diesen Alias-Information berechnen ...
- Achtung: benutzen wir stets die selben globale Variablen $y_1, y_2, \ldots$ zur Simulation der Parameterübergabe, wird die Information dort notwendig ungenau :-(
- Kennen wir das Gesamtprogramm nicht, müssen wir annehmen, dass jede Referenz, die einer anderen Prozedur bekannt ist, entkommt :-((

3.5 Zusammenfassung

Wir haben jetzt diverse Optimierungen kennen gelernt zur besseren Ausnutzung der Hardware-Gegebenheiten.

Reihenfolge ihrer Anwendung:

- Erst globale Restrukturierungen der Prozeduren/Funktionen sowie der Schleifen für besseres Speicherverhalten ;-)
- Dann lokale Umstrukturierung für optimale Nutzung des Instruktionssatzes und der Prozessor-Parallelität :-)
- Dann Registerverteilung und schließlich
- Peephole-Optimierung für den letzten Schliff ...

Funktionen:	Endrekursion + Inlining Stack-Allokation
Schleifen:	Iterationsverbesserung → if-Distribution → for-Distribution Werte-Caching
Rümpfe:	Life-Range-Splitting Instruktions-Auswahl Instruktions-Anordung mit → Schleifen-Abwicklung → Schleifen-Verschmelzung
Instruktionen:	Register-Verteilung Peephole-Optimierung

4 Optimierung funktionaler Programme

Beispiel:

$$\begin{array}{lll} \textbf{fun}\ \ \mathsf{fac}\ x & = & \textbf{if}\ \ x \leq 1\ \ \textbf{then}\ \ 1 \\ & & \textbf{else}\ \ x \cdot \mathsf{fac}\ (x - 1) \end{array}$$

- Es gibt keine Basis-Blöcke :-(
- Es gibt keine Schleifen :-(
- Viele Funktionen sind rekursiv :-((

Strategien zur Optimierung:

⟹ Verbessere spezielle Ineffizienzen wie:

- Pattern Matching
- Lazy Evaluation (falls vorhanden ;-)
- Indirektionen — Unboxing / Escape-Analyse
- Zwischendatenstrukturen — Deforestation

⟹ Entdecke bzw. erzeuge Schleifen mit Basis-Blöcken :-)

- Endrekursion
- Inlining
- **let**-Floating

Wende dann allgemeine Optimierungs-Techniken an!

... etwa durch Übersetzung nach C ;-)

Achtung:

Wir benötigen neue Programmanalyse-Techniken, um Informationen über funktionale Programme zu sammeln.

Beispiel: Inlining

$$
\begin{aligned}
\textbf{fun}\ \ \mathsf{max}\ (x, y) &= \ \ \textbf{if}\ \ x > y\ \ \textbf{then}\ \ x \\
&\ \ \ \textbf{else}\ \ y \\
\textbf{fun}\ \ \mathsf{abs}\ z &= \ \mathsf{max}\ (z, -z)
\end{aligned}
$$

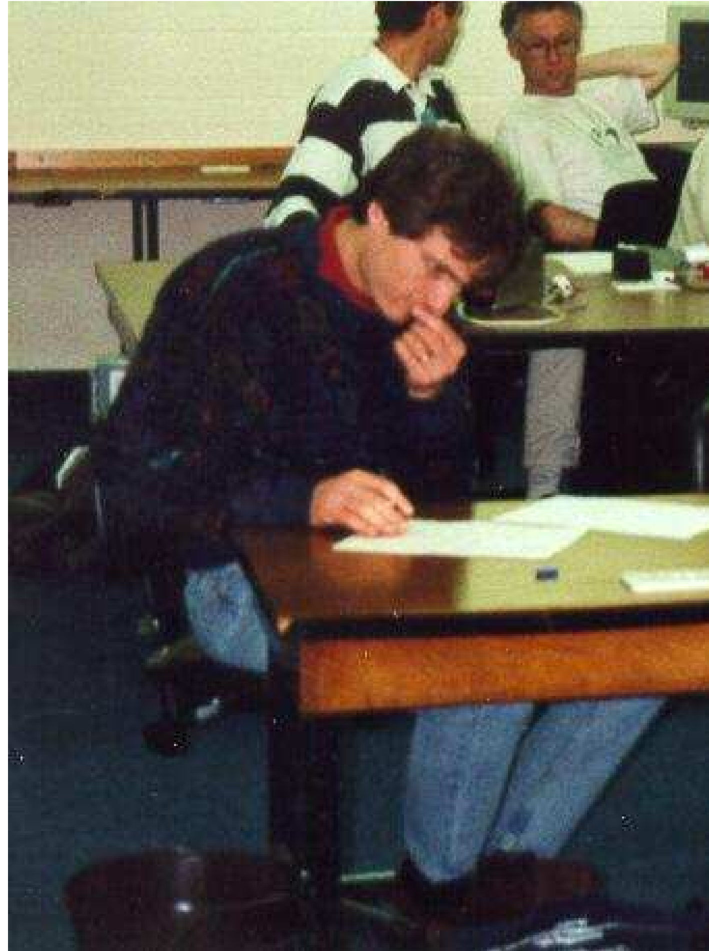
Als Ergebnis der Optimierung erwarten wir ...

$$
\begin{array}{lcl}
\textbf{fun} \;\; \textsf{max}\,(x, y) & = & \textbf{if} \;\; x > y \;\; \textbf{then} \;\; x \\
 & & \textbf{else} \;\; y \\
\textbf{fun} \;\; \textsf{abs}\; z & = & \textbf{let} \quad \textbf{val} \;\; x = z \\
 & & \qquad\;\; \textbf{val} \;\; y = -z \\
 & & \textbf{in} \quad \boxed{\begin{array}{l} \textbf{if} \;\; x > y \;\; \textbf{then} \;\; x \\ \textbf{else} \;\; y \end{array}} \\
 & & \textbf{end}
\end{array}
$$

Diskussion:

max ist zuerstmal nur ein Name. Wirt müssen herausfinden, welchen Wert er zur Laufzeit haben kann

$\Longrightarrow$ Wert-Analyse erforderlich !!



Nevin Heintze im australischen Team
des Prolog-Programmier-Wettbewerbs, 1998

Das ganze Bild:

4.1 Eine einfache Zwischensprache

Zur Vereinfachung betrachten wir:

$$
\begin{array}{lcl}
v & ::= & b \mid (x_1, \ldots, x_k) \mid c\ x \mid \mathbf{fn}\ x \Rightarrow e \\
e & ::= & v \mid (x_1\ x_2) \mid (\Box_1\ x) \mid (x_1\ \Box_2\ x_2) \mid \\
 & & \mathbf{let}\ x_1 = e_1\ \ldots\ x_k = e_k\ \mathbf{in}\ e_0\ \mathbf{end} \mid \\
 & & \mathbf{letrec}\ x_1 = e_1\ \ldots\ x_k = e_k\ \mathbf{in}\ e_0\ \mathbf{end} \mid \\
 & & \mathbf{case}\ x\ \mathbf{of}\ p_1 :\ e_1\ \mid \ldots \mid\ p_k :\ e_k\ \mathbf{end} \\
p & ::= & v \mid x \mid c\ x \mid (x_1, \ldots, x_k)
\end{array}
$$

wobei b eine Konstante ist, x eine Variable, c ein (Daten-)Konstruktor und $\Box_i$ i-stellige Operatoren sind.

Diskussion:

- Konstruktoren und Funktionen sind stets ein-stellig. Dafür gibt es explizite Tupel :-)
- **if**-Ausdrücke und Fall-Unterscheidung in Funktions-Definitionen wird auf **case**-Ausdrücke zurückgeführt.
- In Fall-Unterscheidungen sind nur einfache Muster erlaubt.

  $\Longrightarrow$ Komplizierte Muster müssen zerlegt werden ...
- **let**-Definitionen entsprechen Basis-Blöcken :-)
- Typ-Annotationen an Variablen, Mustern oder Ausdrücken könnten weitere nützliche Informationen enthalten — wir verzichten aber drauf :-)

... im Beispiel:

Die Definition von max sieht dann so aus:

$$
\begin{array}{lll}
\mathsf{max} & = & \mathbf{fn}\ x \Rightarrow \quad \mathbf{case}\ x\ \mathbf{of}\ (x_1, x_2) \ : \\
 & & \qquad\qquad \mathbf{let}\ z = x_1 < x_2 \\
 & & \qquad\qquad \mathbf{in}\ \ \mathbf{case}\ z \\
 & & \qquad\qquad\qquad \mathbf{of}\ \ \mathsf{True}\ :\ \ x_2 \\
 & & \qquad\qquad\qquad\ \ \mid\ \ \mathsf{False}\ :\ \ x_1 \\
 & & \qquad\qquad\qquad \mathbf{end} \\
 & & \qquad\qquad \mathbf{end} \\
 & & \quad \mathbf{end}
\end{array}
$$

Entsprechend haben wir für abs :

$$\begin{aligned} \mathsf{abs} \;=\; \mathbf{fn}\; x \Rightarrow \;\; & \mathbf{let}\; z_1 = -x \\ & \quad\;\; z_2 = (x, z_1) \\ & \mathbf{in}\; (\mathsf{max}\; z_2) \\ & \mathbf{end} \end{aligned}$$

4.2 Eine einfache Wert-Analyse

Idee:

Für jeden Teilausdruck e sammeln wir die Menge $[\![e]\!]^\sharp$ der möglichen Werte von e ...

Sei V die Menge der vorkommenden Konstanten (-Klassen), Konstruktor-Anwendungen und Funktionen. Dann wählen wir als vollständigen Verband natürlich:

$$\mathbb{V} = 2^V$$

Wir stellen wir ein Ungleichungs-System auf:

- Ist e ein Wert d.h. von der Form: $b, c\,x, (x_1, \ldots, x_k)$ oder $\textbf{fn}\,x \Rightarrow e$ erzeugen wir:

$$[\![e]\!]^\sharp \supseteq \{e\}$$

- Ist $e \equiv (x_1\ x_2)$ und $f \equiv \textbf{fn}\ x \Rightarrow e_1$, dann

$$[\![e]\!]^\sharp \supseteq (f \in [\![x_1]\!]^\sharp)\ ?\ [\![e_1]\!]^\sharp : \emptyset$$
$$[\![x]\!]^\sharp \supseteq (f \in [\![x_1]\!]^\sharp)\ ?\ [\![x_2]\!]^\sharp : \emptyset$$

...

- int-Werte, die Operatoren zurück liefern, approximieren wir z.B. durch eine Konstante int.

  Operatoren, die Boolesche Werte liefern, liefern z.B. {True, False} :-)

- Ist $e \equiv \textbf{let}\ x_1 = e_1 \ldots x_k = e_k\ \textbf{in}\ e_0\ \textbf{end}$. Dann erzeugen wir:

$$
\begin{array}{lll}
[\![x_i]\!]^\sharp & \supseteq & [\![e_i]\!]^\sharp \\
[\![e]\!]^\sharp & \supseteq & [\![e_0]\!]^\sharp
\end{array}
$$

- Analog für $e \equiv \textbf{letrec}\ x_1 = e_1 \ldots x_k = e_k\ \textbf{in}\ e_0\ \textbf{end}$:

$$
\begin{array}{lll}
[\![x_i]\!]^\sharp & \supseteq & [\![e_i]\!]^\sharp \\
[\![e]\!]^\sharp & \supseteq & [\![e_0]\!]^\sharp
\end{array}
$$

- Sei $e \equiv \textbf{case}\ x\ \textbf{of}\ p_1\ :\ e_1 \mid \ldots \mid p_k\ :\ e_k\ \textbf{end}$.
  Dann erzeugen wir für $p_i \equiv b$,

$$[\![e]\!]^\sharp \supseteq (b \in [\![x]\!]^\sharp) \,?\, [\![e_i]\!]^\sharp \,:\, \emptyset$$

Ist $p_i \equiv c\,y$ und $v \equiv c\,z$ ein Wert, dann

$$\begin{array}{lll} [\![e]\!]^\sharp & \supseteq & (v \in [\![x]\!]^\sharp) \,?\, [\![e_i]\!]^\sharp \,:\, \emptyset \\ [\![y]\!]^\sharp & \supseteq & (v \in [\![x]\!]^\sharp) \,?\, [\![z]\!]^\sharp \,:\, \emptyset \end{array}$$

Ist $p_i \equiv (y_1, \ldots, y_k)$ und $v \equiv (z_1, \ldots, z_k)$ ein Wert, dann

$$\begin{array}{lll} [\![e]\!]^\sharp & \supseteq & (v \in [\![x]\!]^\sharp) \,?\, [\![e_i]\!]^\sharp \,:\, \emptyset \\ [\![y_j]\!]^\sharp & \supseteq & (v \in [\![x]\!]^\sharp) \,?\, [\![z_j]\!]^\sharp \,:\, \emptyset \end{array}$$

Ist $p_i \equiv y$, dann

$$\begin{array}{lll} [\![e]\!]^\sharp & \supseteq & [\![e_i]\!]^\sharp \\ [\![y]\!]^\sharp & \supseteq & [\![x]\!]^\sharp \end{array}$$

4.3 Eine operationelle Semantik

Idee:

Wir konstruieren eine Big-Step operationelle Semantik, die Ausdrücke auswertet :-)

Konfigurationen:

$$\begin{array}{lll} c & ::= & (e, env) \\ vc & ::= & (v, env) \\ env & ::= & \{x_1 \mapsto vc_1, \ldots\} \end{array}$$

Werte sind Konfigurationen, in denen der Ausdruck von der Form: $b, c\, x, (x_1, \ldots, x_k)$ oder $\mathbf{fn}\, x \Rightarrow e$ ist :-)

Umgebungen enthalten nur Werte :-))

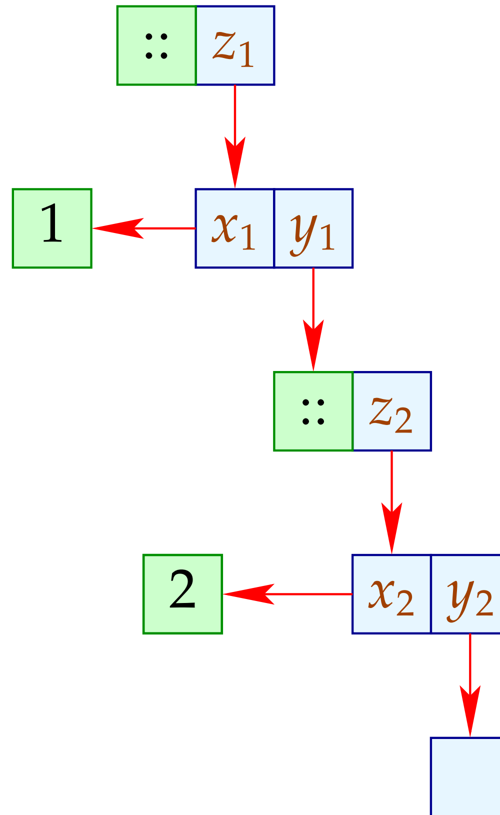
Beispiele für Werte:

$$
\begin{array}{lll}
1 & : & (1, \emptyset) \\
c\,1 & : & (c\ x, \{x \mapsto (1, \emptyset)\}) \\
[1,2] & : & (::\ z_1, \{z_1 \mapsto \\
& & \quad ((x_1, y_1), \{x_1 \mapsto (1, \emptyset), \\
& & \qquad y_1 \mapsto (::\ z_2, \{z_2 \mapsto \\
& & \qquad\quad (x_2, y_2), \{x_2 \mapsto (2, \emptyset), \\
& & \qquad\qquad y_2 \mapsto ((), \emptyset)\})\})\})\})
\end{array}
$$

Werte sehen etwas merkwürdig aus :-)

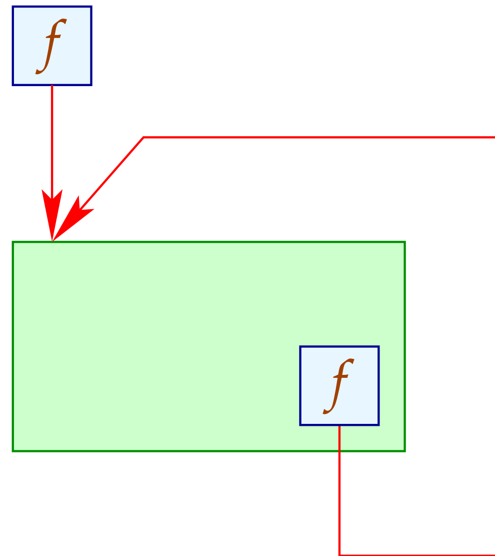
Der Grund ist, dass wir Substitutionen nie ausführen :-)

Alternativ können wir uns die Variablen in den Umgebungen als Speicherzellen vorstellen ...



Achtung:

Rekursive Funktionen führen zu zyklischen Verweis-Strukturen ;-)



Auswege:

- Rekursive Funktionen werden auf dem Toplevel definiert :-)
- Lokale Rekursive Funktionen sind stets nur selbst rekursiv.

  Für diese führen wir einen neuen Operator fix ein ...

Aus: $\textbf{letrec}\ x_1 = e_1\ \textbf{in}\ e_0\ \textbf{end}$

wird: $\textbf{let}\ x_1 = \mathsf{fix}\,(x_1, e_1)\ \textbf{in}\ e_0\ \textbf{end}$

Beispiel: Die append-Funktion

Betrachten wir die Konkatenation von zwei Listen. In ML schreiben wir einfach:

$$\begin{array}{llcl} \textbf{fun} & \mathsf{app}\ [\,] & = & \textbf{fn}\ y \Rightarrow y \\ \mid & \mathsf{app}\ (x :: xs) & = & \textbf{fn}\ y \Rightarrow x :: \mathsf{app}\ xs\ y \end{array}$$

In unserer eingeschränkten Zwischensprache sieht das etwas detaillierter aus :-)

$$
\begin{array}{rcl}
\mathsf{app} & = & \mathsf{fix}\,(\mathsf{app},\ \textbf{fn}\ x \Rightarrow \textbf{case}\ x \\
& & \quad \textbf{of}\ \ [\,]\ :\ \ \textbf{fn}\ y\ \Rightarrow\ y \\
& & \quad \mid\ \ ::\ z :\ \ \textbf{case}\ z\ \textbf{of}\ (x_1, x_2)\ :\ \ \textbf{fn}\ y\ \Rightarrow \\
& & \qquad\qquad \textbf{let}\ \ a_1\ =\ \mathsf{app}\ x_2 \\
& & \qquad\qquad\qquad a_2\ =\ a_1\ y \\
& & \qquad\qquad\qquad z_1\ =\ (x_1, a_2) \\
& & \qquad\qquad \textbf{in}\ \ ::\ z_1 \\
& & \qquad\qquad \textbf{end} \\
& & \qquad \textbf{end} \\
& & \quad \textbf{end}\ )
\end{array}
$$

Die Big-Step Semantik gibt Regeln an, zu welchem Wert sich eine Konfiguration ausrechnen lässt ...