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 $\rightarrow$ if-Distribution $\rightarrow$ for-Distribution Werte-Caching
Rümpfe:	Life-Range-Splitting Instruktions-Auswahl Instruktions-Anordung mit $\rightarrow$ Schleifen-Abwicklung $\rightarrow$ Schleifen-Verschmelzung
Instruktionen:	Register-Verteilung Peephole-Optimierung

4 Optimierung funktionaler Programme

Beispiel:

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

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

Strategien zur Optimierung:

$\Longrightarrow$ Verbessere spezielle Ineffizienzen wie:

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

$\Longrightarrow$ 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

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

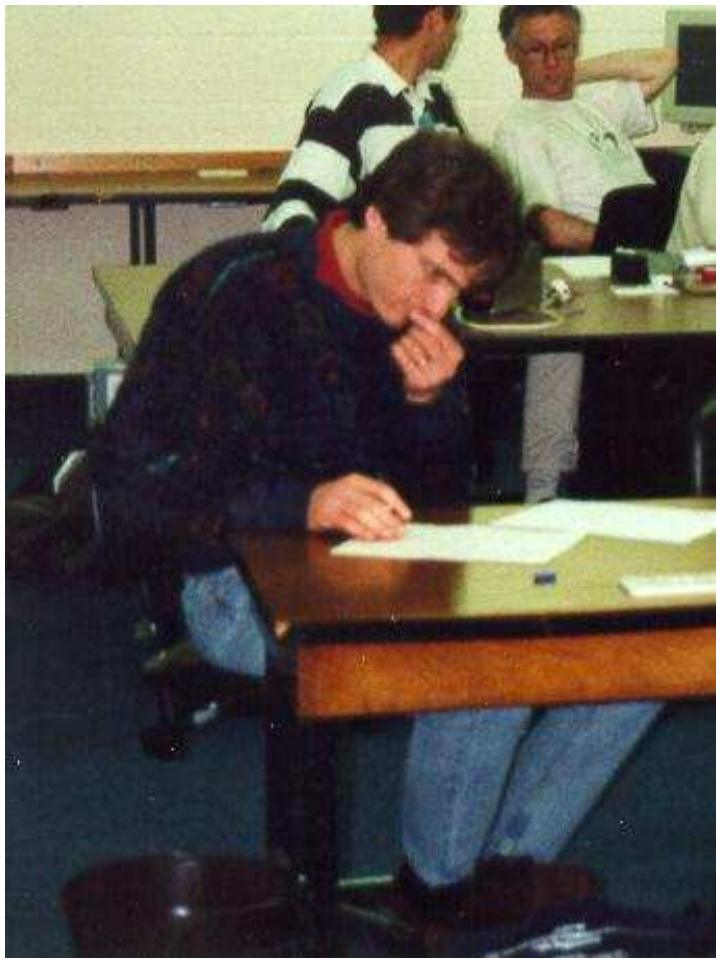
Als Ergebnis der Optimierung erwarten wir ...

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

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:

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

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:

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

Entsprechend haben wir für abs :

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

Idee für eine Analyse:

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 $\mathbf{fn}\,x \;\Rightarrow\; e$ erzeugen wir:

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

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

$$\begin{array}{rcl} [\![e]\!]^\sharp & \supseteq & (f \in [\![x_1]\!]^\sharp)\,?\,[\![e_1]\!]^\sharp \;:\; \emptyset \\ [\![x]\!]^\sharp & \supseteq & (f \in [\![x_1]\!]^\sharp)\,?\,[\![x_2]\!]^\sharp \;:\; \emptyset \end{array}$$

...

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

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

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

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

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

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

- 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

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