Elimination von Funktionen höherer Ordnung

Torsten Landschoff, betreut durch Prof. Dr. Hanus

7. Juni 2004

Inhaltsverzeichnis

1 Funktionen höherer Ordnung

Funktionen höherer Ordnung sind solche, die entweder eine Funktion als Rückgabewert liefern oder aber eine Funktion als Parameter akzeptieren. Damit helfen Sie dem Programmierer, Problemlösungen abstrakter anzugeben, auf der anderen Seite jedoch erschweren sie die Generierung von effizientem Code.

Zum Beispiel berechnet folgende Haskell-Funktion die Summe der Zahlen 1 bis n:

```
sumTo n = if n > 0 then n + sumTo (n-1) else n
```

Genauso tut das auch die folgende Funktion – man vergleiche die Lesbarkeit:

```
sumTo n = foldr (+) 0 [1..n]
```

Ein sehr schönes Beispiel für Funktionen höherer Ordnung ist auch die Anwendung einer Funktion auf jeden Eintrag einer Datenstruktur. So sind folgende Funktionen äquivalent:

```
inclist v xs = map (+v) xs
inclist' v xs =
  if xs == [] then xs else (v + head xs) : inclist' v (tail xs)
```

Dabei hat die erste Fassung mit `map` neben der kurzen Darstellung noch den Vorteil, dass sie auch leicht auf andere Datenstrukturen angepaßt werden kann, indem man statt `map` eine entsprechende Funktion aufruft. Mit den Sprachmitteln von Haskell ist es übrigens möglich, die zweite Variante der Funktion wesentlich prägnanter zu formulieren, aber in Lisp-ähnlichen und in prozeduralen Sprachen ist diese Form durchaus realistisch.

Neben den angegebenen Beispielen, in denen jeweils eine Funktion als Parameter übergeben wird, gibt es auch noch das Gegenteil, nämlich, dass eine Funktion als das Ergebnis eines Funktionsaufrufes zurückgegeben wird. Ein triviales Beispiel wäre folgende Funktion:

```
inc x = \y -> y + x
```

Funktionen höherer Ordnung werden auch häufig als Abbildung verwendet, die zur Laufzeit berechnet wird, beispielsweise aus der Analyse eines mathematischen Ausdruckes oder als Anpassung einer bekannten Funktion. Dazu als Beispiel die Implementierung eines assoziativen Speichers mit Funktionen höherer Ordnung:

```
empty x = error "Leeres Array"
get array key = array key
set array key value = \x -> if x == key then value else array x
```

Die Funktion `empty` eine einstellige Funktion, die für jeden Parameter einen Fehler erzeugt (entsprechend einem leeren Array), die Funktion `get` ruft einfach eine angegebene Funktion mit dem Schlüssel als Parameter auf, um den gespeicherten Wert zu finden, und die Funktion `set` liefert schließlich eine Funktion, die für den angegebenen Schlüssel einen bestimmten Wert liefert, und sonst die alte Abbildung aufruft, also einen neuen Wert abspeichert.

1.1 Motivation

Während Funktionen höherer Ordnung hilfreich sind für die Lesbarkeit und Wiederverwendbarkeit des Quellcodes, haben sie aus der Sicht des Compilers einige Nachteile:

1. Sie erschweren die Optimierung, insbesondere von trivialen Funktionen. Zum Beispiel kann die Addition gewöhnlich zu einem Maschinenbefehl optimiert werden, wird sie dagegen als Funktionsparameter übergeben, so muß der Compiler jeweils einen Funktionsaufruf generieren, der zeitintensiver ist, als die eigentliche Rechenoperation.

2. Bei Angabe eines Lambda-Ausdruckes muß jeweils die gesamte Umgebung festgehalten werden, um eine spätere Auswertung zu ermöglichen (z.B. werden in der obigen `set`-Funktion die Werte von `key` und `value` festgehalten, genauso bei `inc` der Wert von `x`). Daher bedingt die Verwendung von Funktionen höherer Ordnung ein Laufzeitsystem mit Garbage Collection, die diese Umgebungen nach der letzten Nutzung wieder freigibt.

1.2 Umformung einiger trivialer Beispiele

Wie schon erwähnt, möchte man Funktionen höherer Ordnung grundsätzlich vermeiden, soweit das möglich ist. Die Idee dabei ist, den Funktionsrumpf an der Stelle einzusetzen, wo die Funktion höherer Ordnung aufgerufen wird. Man betrachte etwa folgende Umformung:

$$(\text{inc } 5)\ 6 \Rightarrow (\backslash y \rightarrow y + 5)\ 6 \Rightarrow 6 + 5$$

Hier wird im ersten Schritt die Definition von `inc` ein- und dabei der Wert für x textuell ersetzt, im zweiten Schritt wird dann der Lambda-Ausdruck

eliminiert – da der eine Parameter direkt angegeben ist, kann er einfach eingesetzt werden.

Für das folgende Beispiel wird die Funktion `eval` verwendet:

```
eval f = \x -> f x
```

Damit ergibt sich folgende Ersetzung:

$$(\text{eval } (\text{inc } 1))\ 7 \Rightarrow (\backslash x \rightarrow (\text{inc } 1)\ x)\ 7 \Rightarrow (\backslash x \rightarrow x + 1)\ 7 \Rightarrow 7 + 1$$

Hier wird zunächst die Definition von `eval` angewendet und dabei für `f` der „Wert“ `inc 1` eingesetzt. Dann wird der Aufruf von `inc` ausgefaltet und am Ende der Lambda-Ausdruck reduziert.

1.3 Typüberprüfung

Bisher betrachtete Beispiele wurden immer so ausführlich geklammert, dass klar erkennbar war, wo ein Ausdruck eine Funktion zum Ergebnis hatte. Schwierig wird es z.B. mit dem Ausdruck `inc 5 7`. Vom bloßen Hinsehen kann man nicht mehr erkennen, ob hier eine Funktion höherer Ordnung im Spiel ist.

Für einen Haskell-Compiler ist es allerdings kein Problem, festzustellen, was hier passiert, da er während der Programmanalyse mittels des Algorithmus' von Hindley-Milner auf die Typen aller Ausdrücke schließt. Damit steht nach der Definition fest, dass `inc` vom Typ `Num a => a -> a -> a` ist (Haskell-Notation) – und damit eine Funktion höherer Ordnung.

Genaugenommen ist jede Funktion, die mehr als einen Parameter bekommt, in Haskell tatsächlich eine Funktion höherer Ordnung, denn die Definition `add a b = a + b` ist in Haskell nur eine abkürzende Schreibweise für die Definition `add a = \b -> a + b`.

2 Idee des Algorithmus'

Das vorgestellte Verfahren geht in zwei Durchläufen vor: Im ersten Durchlauf wird bottom-up die Verwendungen von Rückgabewerten als Funktion entfernt. Im zweiten Schritt wird das gesamte Programm noch einmal bearbeitet. Diesmal werden (wiederum bottom-up) Aufrufe mit Funktionen als Parametern entfernt.

Bottom-up steht hier dafür, dass der Rumpf einer Funktion f, die von der Funktion g aufgerufen wird, zuerst betrachtet wird. Das ist schon deswegen sinnvoll, weil sonst beim potenziellen Einsetzen der Definition von f in g die gesamte Bearbeitung von f mehrmals geleistet würde.

3 Verwendete Notation

Diese Ausarbeitung stellt den Versuch dar, den in [1] vorgestellten Algorithmus anschaulich darzustellen. Wer am Korrektheitsbeweis und einer mathematisch korrekten Darstellung interessiert ist, sei an die Vorlage verwiesen.

3.1 Programmiersprache

Die hier aufgeführten Beispiele sind aus praktischen Gründen meistens in Haskell angegeben. Es wäre jedoch bei weitem zu aufwändig, den Algorithmus für alle Sprachmerkmale von Haskell anzugeben. Die Darstellung des Verfahrens beschränkt sich daher auf folgende Haskell-ähnliche Sprache. Es läßt sich jedoch prinzipiell genauso auf den ganzen Sprachschatz von Haskell ausdehnen.

$$
\begin{array}{rcl}
P & ::= & \text{declare } X \text{ in } P \mid t \\
X & ::= & D \mid F \\
D & ::= & \text{data } C\ \alpha_1 \ldots \alpha_n = c_1\ \tau_{1_1} \ldots \tau_{1_j} | \ldots | c_m\ \tau_{m_1} \ldots \tau_{m_j} \\
F & ::= & \{E_1; \ldots ; E_n\} \\
E & ::= & f\ v_1 \ldots v_n = t \\
t & ::= & op\ t_1 \ldots t_n \\
op & ::= & v \mid f \mid c \mid \backslash v_1 \ldots v_n \rightarrow t \mid \text{case } t \text{ of } \{p_1 \rightarrow t_1; \ldots ; p_n \rightarrow t_n\} \\
p & ::= & v \mid c\ p_1 \ldots p_j \\
\tau & ::= & \alpha \mid C\ \tau_1 \ldots \tau_n \mid \tau_1 \rightarrow \tau_2
\end{array}
$$

Dabei steht v für eine Variable, f für einen Funktionsbezeichner, c für einen Datenkonstruktor, C für einen Typkonstruktor und α für eine Typvariable.

3.2 Ersetzungsregeln

Der dargestellte Algorithmus basiert auf Termersetzungsregeln, die hier in der Form

$$[\![f\ a\ b]\!] \Rightarrow g\ a\ [\![b]\!]$$

dargestellt werden. Dabei bedeutet $[\![b]\!]$, dass der Ausdruck b noch (rekursiv) analysiert werden muß.

Die verschiedenen Ersetzungsregeln sind nur unter verschiedenen Vorbedingungen anzuwenden und können Seiteneffekte haben. Beides wird hier nur in Prosa angegeben.

Außerdem wird die Notation $t[t_i/v_i]_{i\in\ldots}$ verwendet. Diese Abkürzung steht für den Ausdruck t, in dem t_i für jedes Auftreten von v_i eingesetzt wird.

4 Elimination von Funktionsrückgaben

Die erste Stufe des Algorithmus' soll dafür sorgen, dass Funktionen, die von solchen höherer Ordnung geliefert werden, ausgefaltet werden.

4.1 Eta-Expansion

Die ersten beiden Regeln dienen dem Zweck, für die folgenden Regeln sicherzustellen, dass alle Funktionsanwendungen voll parametrisiert sind. Regel (2) wird angewendet, wenn der Ausdruck t einen Funktionstyp hat. Falls das der Fall ist, wird der Ausdruck in eine Lambda-Abstraktion eingebettet. Wenn z.B. add die zweistellige Addition ist, würde der Ausdruck

$$(\text{add } x) \quad \text{zu} \quad (\backslash v_1 \rightarrow \text{add } x\ v_1)$$

expandiert – der Hauptteil des Algorithmus wird dann nur noch auf den Definitionsteil des Lambda-Ausdrucks angewendet:

$$[\![t]\!] \Rightarrow \backslash v_1 ... v_n \rightarrow [\![t\ v_1 ... v_n]\!] \tag{2}$$

Dabei ist n die Funktionsordnung des Ausdruckes t, also die Anzahl der Parameter, die t für die komplette Auswertung benötigt.

Außerdem gibt es den Sonderfall der „reinen“ Lambda-Ausdrücke. Obige Expansion angewendet auf

$$(\backslash x\ y \rightarrow \text{add } x\ y) \quad \text{ergibt} \quad (\backslash v_1\ v_2 \rightarrow (\backslash x\ y \rightarrow \text{add } x\ y)\ v_1\ v_2))$$

Dafür ist Regel (1) zuständig. Wird nämlich der Algorithmus auf reine Lambda-Ausdrücke angewendet, wird er einfach für den Rumpf der Lambda-Abstraktion rekursiv aufgerufen.

$$[\![\backslash v_1 ... v_n \rightarrow tf]\!] \Rightarrow \backslash v_1 ... v_n \rightarrow [\![tf]\!] \tag{1}$$

Für das obige Beispiel wird folgende Ersetzung ausgeführt:

$$[\![\backslash x\ y \rightarrow \text{add } x\ y]\!] \quad \Rightarrow \quad \backslash x\ y \rightarrow [\![\text{add } x\ y]\!]$$

4.2 Variablenanwendung

Wird in einer Variablen v eine Funktion gespeichert und diese über die Variable aufgerufen, so ist es ohne weitere Informationen nicht möglich, die evtl. verwendeten Funktionen höherer Ordnung zu reduzieren, denn dazu müßte der Laufzeitwert von v zur Übersetzungszeit bekannt sein. Daher kann der Algorithmus nur die Teilausdrücke rekursiv bearbeiten. Das wird von Regel (3) erledigt:

$$[\![v\ t_1 ... t_n]\!] \Rightarrow v\ [\![t_1]\!] ... [\![t_n]\!] \tag{3}$$

4.3 Konstruktoranwendung

Ein Konstrukturausdruck liefert schon einen nicht-funktionalen Wert, daher kann auch an einem solchen Ausdruck nichts optimiert werden. Analog zu oben wird die Optimierung nur rekursiv auf den Parametern ausgeführt:

$$[\![c\ t_1 ... t_n]\!] \Rightarrow c\ [\![t_1]\!] ... [\![t_n]\!] \tag{4}$$

4.4 Fallunterscheidung

Bei einer Fallunterscheidung reicht es unter Umständen nicht, nur rekursiv die vorkommenden Terme zu optimieren, wie folgendes Beispiel verdeutlicht:

$$\text{case } dir \text{ of } \{\text{left} \rightarrow \text{inc}(-1);\ \text{right} \rightarrow \text{inc } 1\}\ pos$$

Folgende Ersetzung ermöglicht hier die Eliminierung der Funktion höherer Ordnung:

$$\begin{array}{ll} & [\![\text{case } dir \text{ of } \{\text{left} \rightarrow \text{inc}(-1);\ \text{right} \rightarrow \text{inc } 1\}\ pos]\!] \\ \Rightarrow & \text{case } [\![dir]\!] \text{ of } \{\text{left} \rightarrow [\![\text{inc}(-1)\ pos]\!];\ \text{right} \rightarrow [\![\text{inc } 1\ pos]\!]\} \\ \Rightarrow^* & \text{case } dir \text{ of } \{\text{left} \rightarrow (-1) + pos;\ \text{right} \rightarrow 1 + pos\} \end{array}$$

Es ist also nötig, die Terme, auf die der case-Ausdruck angewendet wird, in die verschiedenen Zweige hereinzuziehen. Das leistet die folgende Transformation:

$$[\![\text{case } t \text{ of } \{p_i \rightarrow tf_i\}_{i \in M} t_1 ... t_n]\!] \Rightarrow \text{case } [\![t]\!] \text{ of } \{p_i \rightarrow [\![tf_i\ t_1 ... t_n]\!]\}_{i \in M} \tag{5}$$

4.5 Lambda-Anwendung

Da wegen der Eta-Expansion sicher ist, dass alle Lambda-Anwendungen voll parametrisiert sind, ist es möglich, die Parameter textuell in den Rumpf des Lambda-Ausdruckes einzusetzen. Dies wird von folgender Regel geleistet:

$$[\![(\backslash v_1 ... v_m \rightarrow tf)\ t_1 ... t_n]\!] \Rightarrow [\![tf[t_i/v_i]_{i \in 1..m}\ t_{m+1} ... t_n]\!] \tag{6}$$

Diese Ersetzung ist dann von Nachteil, wenn ein Parameter t_i einen komplizierten Ausdruck darstellt und mehrfach im Lambda-Rumpf vorkommt – der Wert müßte mehrfach berechnet werden. Kommt also ein solcher Parameter vor, so wird der Algorithmus stattdessen folgende Ersetzung durchführen:

$$[\![(\backslash v_1 ... v_m \rightarrow tf)\ t_1 ... t_n]\!] \Rightarrow f_{\text{NEW}}\ [\![t_1]\!] ... [\![t_n]\!]\ \vec{w} \tag{7}$$

$\vec{w}$ bezeichnet hier die freien Variablen im Ausdruck tf. Als Nebeneffekt wird eine neue Funktion f_{NEW} wie folgt definiert:

$$f_{\text{NEW}}\ v_1 ... v_n\ \vec{w} = [\![tf\ v_{m+1} ... v_n\ \vec{w}]\!]$$

Dass dies nützlich ist, zeigt folgendes Beispiel:

```
(\x -> x + x) (length [1..1000])
```

Durch reines Einsetzen wird daraus der Ausdruck

```
(length [1..1000]) + (length [1..1000])
```

Verwendet man dagegen die Regel (7) und führt eine neue Funktion ein, erhält man dagegen folgenden Haskell-Code, der mit Hugs auch tatsächlich um den Faktor 2 schneller läuft:

```
let f x = x + x in f (length [1..1000])
```

Streng genommen hat das Entfernen dieser Lambda-Anweisungen nicht direkt etwas mit Funktionen höherer Ordnung zu tun, es könnte aber sein, dass ein Lambda-Ausdruck wiederum eine Funktion liefert, und dann ist das Einsetzen notwendig. Und in jedem Fall ist es gewöhnlich für den Compiler günstiger, Funktionen einzuführen, statt jeweils Lambda-Ausdrücke zu verwenden (Stichwort Vermeidung von Closures).

4.6 Funktions-Anwendung

Im einfachsten Fall wird die Funktion einfach mit der Anzahl Parametern aufgerufen wird, für die sie deklariert wurde. Daran kann nichts optimiert werden, es handelt sich einfach um einen ordinären Funktionsaufruf:

$$[\![f\ t_1 ... t_n]\!] \Rightarrow f\ [\![t_1]\!] ... [\![t_n]\!] \tag{8}$$

Folgen den notwendigen Parametern noch weitere, so folgt, dass die Funktion eine Funktion als Rückgabewert liefert und diese mit den überzähligen Parametern ausgewertet wird. Um diese Funktion höherer Ordnung zu eliminieren, wird folgende Regel angewendet:

$$[\![f\ t_1 ... t_n]\!] \Rightarrow f'_{\text{NEW}} [\![t_1]\!] ... [\![t_n]\!] \tag{9}$$

Dabei wird eine neue Funktion f'_{NEW} eingeführt, die wie folgt definiert ist (tf ist dabei der Rumpf der Funktion f):

$$f'_{\text{NEW}} v_1 ... v_n = [\![tf\ v_{a+1} ... v_n]\!]$$

Wurde für die gleichen Parameter von f schonmal eine neue Funktion eingeführt, so wird diese einfach erneut aufgerufen, anstatt die Expandierung nochmal auszuführen.

Da das der Kern des ganzen Verfahrens ist, sei hier ein ausführlicheres Beispiel angegeben. Die Funktion collect sei wir folgt definiert:

$$\text{collect}\ f\ s\ = \backslash x \rightarrow \text{case}\ x\ \text{of}\ \{0 \rightarrow s;\ n+1 \rightarrow f\ x\ (\text{collect}\ f\ s\ n)\}$$

Dann kann der Ausdruck (collect $(*)$ 1 5) wie folgt optimiert werden:

$$\begin{array}{ll} & [\![\text{collect}\ (*)\ 1\ 5]\!] \\ \overset{(9)}{\Rightarrow} & \text{collect}'[\![(*)]\!][\![1]\!][\![5]\!] \\ \overset{*}{\Rightarrow} & \text{collect}'\ (*)\ 1\ 5 \end{array}$$

Dabei wird die Funktion collect$'$ wie folgt eingeführt:

$$
\begin{aligned}
\text{collect}'\ f\ s\ y &= [\![\backslash x \to \text{case } x \text{ of } \{0 \to s;\ n+1 \to f\ x\ (\text{collect } f\ s\ n)\}\ y]\!] \\
&\overset{(6)}{=} [\![\text{case } y \text{ of } \{0 \to s;\ n+1 \to f\ y\ (\text{collect } f\ s\ n)\}]\!] \\
&\overset{(5)}{=} \text{case } [\![y]\!] \text{ of } \{0 \to [\![s]\!];\ n+1 \to [\![f\ y\ (\text{collect } f\ s\ n)]\!]\} \\
&\overset{(3)}{=} \text{case } y \text{ of } \{0 \to [\![s]\!];\ n+1 \to [\![f\ y\ (\text{collect } f\ s\ n)]\!]\} \\
&\overset{(3)}{=} \text{case } y \text{ of } \{0 \to s;\ n+1 \to [\![f\ y\ (\text{collect } f\ s\ n)]\!]\} \\
&\overset{(3)}{=} \text{case } y \text{ of } \{0 \to s;\ n+1 \to f\ [\![y]\!]\ ([\![\text{collect } f\ s\ n]\!])\} \\
&\overset{(3)}{=} \text{case } y \text{ of } \{0 \to s;\ n+1 \to f\ y\ ([\![\text{collect } f\ s\ n]\!])\} \\
&\overset{(9)}{=} \text{case } y \text{ of } \{0 \to s;\ n+1 \to f\ y\ (\text{collect}'\ [\![f]\!]\ [\![s]\!]\ [\![n]\!])\} \\
&\overset{*}{=} \text{case } y \text{ of } \{0 \to s;\ n+1 \to f\ y\ (\text{collect}'\ f\ s\ n)\}
\end{aligned}
$$

Man bemerke, dass beim vorletzten Schritt festgestellt wird, dass es schon eine Funktion collect$'$ gibt, die einem Aufruf von collect mit 3 Parametern entspricht. Ansonsten würde jetzt eine weitere Funktion eingeführt, die wiederum genau den gleichen Rumpf bekäme, wodurch wiederum eine weiter Funktion eingeführt wird ad infinitum.

Die Rückgabe von Funktionen haben wir in diesem Beispiel also jetzt behoben, allerdings kommt noch eine Funktion als Parameter vor.

4.7 Regeln

Hier noch einmal die Regeln für diese erste Hälfte des Verfahrens im Überblick. Voraussetzungen und Nebenwirkungen der Anwendung werden in den vorangegangenen Abschnitten beschrieben.

$$
\begin{aligned}
[\![\backslash v_1 ... v_n \to tf]\!] &\Rightarrow \backslash v_1 ... v_n \to [\![tf]\!] && (1)\\
[\![t]\!] &\Rightarrow \backslash v_1 ... v_n \to [\![t\ v_1 ... v_n]\!] && (2)\\
[\![v\ t_1 ... t_n]\!] &\Rightarrow v\ [\![t_1]\!] ... [\![t_n]\!] && (3)\\
[\![c\ t_1 ... t_n]\!] &\Rightarrow c\ [\![t_1]\!] ... [\![t_n]\!] && (4)\\
[\![\text{case } t \text{ of } \{p_i \to tf_i\}_{i \in M} t_1 ... t_n]\!] &\Rightarrow \text{case } [\![t]\!] \text{ of } \{p_i \to [\![tf_i\ t_1 ... t_n]\!]\}_{i \in M} && (5)\\
[\![(\backslash v_1 ... v_m \to tf)\ t_1 ... t_n]\!] &\Rightarrow [\![tf[t_i/v_i]_{i \in 1..m}\ t_{m+1} ... t_n]\!] && (6)\\
[\![(\backslash v_1 ... v_m \to tf)\ t_1 ... t_n]\!] &\Rightarrow f_{\text{NEW}} [\![t_1]\!] ... [\![t_n]\!] && (7)\\
[\![f\ t_1 ... t_n]\!] &\Rightarrow f\ [\![t_1]\!] ... [\![t_n]\!] && (8)\\
[\![f\ t_1 ... t_n]\!] &\Rightarrow f'_{\text{NEW}} [\![t_1]\!] ... [\![t_n]\!] && (9)
\end{aligned}
$$

5 Elimination von Funktionsparametern

Die folgenden Transformationen dienen dazu, im zweiten Durchlauf möglichst viele Funktionsparameter zu eliminieren. Dabei wird angenommen, dass die obigen Transformationen schon ausgeführt wurden und zum Beispiel keine case-Ausdrücke mehr angewendet werden (denn die Parameter wurden ja im ersten Schritt in die verschiedenen Alternativen reingezogen).

Zur Vereinfachung der Darstellung wird angenommen, dass alle Parameter mit Funktionstyp am Ende der Argumentliste stehen. Die Verallgemeinerung besteht einfach in einer Umordnung der Parameter bei der Optimierung.

5.1 Nicht optimierbare Ausdrücke

Die Regeln 1 bis 4 sind für Ausdrücke, die wie oben nicht optimiert werden können, weil sie eine Variablenanwendung, einen Konstruktoraufruf, eine Fallunterscheidung oder einen Lambda-Ausdruck darstellen. Es werden einfach nur die Teilausdrücke rekursiv optimiert:

$$\begin{aligned}
[\![v\ t_1 ... t_n]\!] &\Rightarrow v\ [\![t_1]\!] ... [\![t_n]\!] && (1)\\
[\![c\ t_1 ... t_n]\!] &\Rightarrow c\ [\![t_1]\!] ... [\![t_n]\!] && (2)\\
[\![\text{case } t \text{ of } \{p_i \rightarrow tf_i\}_{i \in 1..n}]\!] &\Rightarrow \text{case } [\![t]\!] \text{ of } \{p_i \rightarrow [\![tf_i]\!]\}_{i \in 1..n} && (3)\\
[\![\backslash v_1 ... v_n \rightarrow t]\!] &\Rightarrow \backslash v_1 ... v_n \rightarrow [\![t]\!] && (4)
\end{aligned}$$

5.2 Funktionsaufrufe

Für die Optimierung von Funktionsaufrufen wird angenommen, dass die Parameter $t_1 ... t_a$ keinen Funktionstyp haben und die Parameter $t_{a+1} ... t_n$ Funktionen sind und keine Variablenausdrücke sind. Ansonsten ist eine Umordnung der Parameter vorzunehmen.

Ist $a = n$, gibt es also keinen Funktionsparameter, der nicht durch eine Variable ausgedrückt ist, so kann nichts optimiert werden, da über die übergebenen Funktionen nichts bekannt ist. In diesem Fall sorgt die Regel 5 dafür, dass die Parameter rekursiv optimiert werden:

$$[\![f\ t_1 ... t_n]\!] \Rightarrow f\ [\![t_1]\!] ... [\![t_n]\!] \quad (5)$$

Ist einer der Funktionsparameter keine Variable, so wird die Regel 6 angewendet:

$$[\![f\ t_1 ... t_n]\!] \Rightarrow f_{\text{NEW}} [\![t_1]\!] ... [\![t_a]\!]\ \vec{w} \quad (6)$$

Gleichzeitig wird eine neue Funktion eingeführt, die wie folgt definiert ist (tf steht für den Rumpf der Funktion f, deren Aufruf substituiert werden soll, und $\vec{w}$ für die freien Variablen, die in tf vorkommen):

$$f_{\text{NEW}}\ v_1 ... v_a\ \vec{w} = [\![tf[t_i/v_i]_{i \in a+1..n}]\!]$$

Diese Definition ist somit eine Spezialisierung von f mit den bekannten Funktionsparametern. Zur Veranschaulichung sei hier nochmal auf die Funktion collect und den Ausdruck (collect $(*)$ 1 5) zurückgegriffen. Durch die Eliminierung von Funktionsrückgaben ergibt sich daraus der neue Ausdruck collect$'$ $(*)$ 1 5 unter folgender Definition von collect$'$:

$$\text{collect}'\ f\ s\ y = \text{case}\ y\ \text{of}\ \{0 \to s;\ n+1 \to f\ y\ (\text{collect}'\ f\ s\ n)\}$$

Die Eliminierung von Funktionsparametern auf collect$'$ $(*)$ 1 5 liefert nun folgende Schritte:

$$\begin{aligned} & [\![\text{collect}'\ (*)\ 1\ 5]\!] \\ \overset{(6)}{\Rightarrow}\ & \text{collect}''\ [\![1]\!]\ [\![5]\!] \\ \overset{*}{\Rightarrow}\ & \text{collect}''\ 1\ 5 \end{aligned}$$

Dabei wird collect$''$, die Spezialisierung von collect$'$ für $f = (*)$, wie folgt eingeführt:

$$\begin{aligned} \text{collect}''\ s\ y &= [\![\backslash x \to \text{case}\ x\ \text{of}\ \{0 \to s;\ n+1 \to x * (\text{collect}'\ (*)\ s\ n)\}\ y]\!] \\ &\overset{(8)}{=} [\![\text{case}\ y\ \text{of}\ \{0 \to s;\ n+1 \to x * (\text{collect}'\ (*)\ s\ n)\}]\!] \\ &\overset{(3)}{=} \text{case}\ [\![y]\!]\ \text{of}\ \{0 \to [\![s]\!];\ n+1 \to [\![x * (\text{collect}'\ (*)\ s\ n)]\!]\} \\ &\overset{(1)}{=} \text{case}\ y\ \text{of}\ \{0 \to [\![s]\!];\ n+1 \to [\![x * (\text{collect}'\ (*)\ s\ n)]\!]\} \\ &\overset{(1)}{=} \text{case}\ y\ \text{of}\ \{0 \to s;\ n+1 \to [\![x * (\text{collect}'\ (*)\ s\ n)]\!]\} \\ &\overset{(5)}{=} \text{case}\ y\ \text{of}\ \{0 \to s;\ n+1 \to x * [\![(\text{collect}'\ (*)\ s\ n)]\!]\} \\ &\overset{(6)}{=} \text{case}\ y\ \text{of}\ \{0 \to s;\ n+1 \to x * [\![(\text{collect}''\ s\ n)]\!]\} \end{aligned}$$

Der letzte Schritt hätte wieder eine neue Spezialisierung von collect$'$ einführen müssen, was offensichtlich zu einer endlosen Rekursion geführt hätte. Aber da collect$''$ schon die Spezialisierung von collect$'$ mit $f = (*)$ ist, wird an der Stelle die schon vorhandene Funktion eingesetzt.

Es gibt noch eine weitere Gefahr bei der Spezialisierung von Funktionen. Und zwar ist es möglich, dass ein Funktionsparameter bei einem rekursiven Aufruf erweitert wird, wie in folgendem Beispiel:

```
acc f = \x -> f (x-1)
id x = x
boom f x = if (f x) == 0 then 1 else (f x)*(boom (acc f) x)
```

Der Ausdruck boom id 5 berechnet nun die Fakultät von 5. Möchte man diesen Ausdruck optimieren, so wird die Funktion boom$'$ eingeführt, was der Spezialisierung von boom mit $f = \text{id}$ entspricht. Im nächsten Schritt wird dann boom$''$ definiert für $f = \text{acc id}$ und so weiter.

5.3 Lambda-Ausdrücke

Lambda-Ausdrücke wurden prinzipiell weitestmöglich im ersten Schritt eliminiert, allerdings konnten reine Lambda-Ausdrücke in Funktionskörpern natürlich bisher nicht eliminiert werden. Beim Ausfalten können diese jetzt wieder auftauchen, wie in obigem Beispiel in collect$'$. Daher werden sie hier erneut mit sehr ähnlichen Regeln behandelt.

Zuerst wird in Regel 7 wieder eine Eta-Expansion ausgeführt, denn der erste Schritt des Algorithmus hat evtl. nicht genug Informationen, da im Lambda-Ausdruck eine als Parameter übergebene Funktion vorkommen kann. Hier steht n' für die Anzahl der zusätzlichen Parameter an, die nötig sind, um den Ausdruck zu einem nicht-funktionalen Typ auszuwerten.

$$[\![(\backslash v_1 ... v_m \to tf)\ t_1 ... t_n]\!] \Rightarrow \backslash w_1 ... w_{n'} \to [\![(\backslash v_1 ... v_m \to tf)\ \vec{t}\ \vec{w}]\!] \qquad (7)$$

Eine Möglichkeit, einen Lambda-Aufruf (und damit Funktionsparameter dieses Aufrufes) zu eliminieren, besteht darin, wie schon in Teil 1 den Lambda-Ausdruck auszufalten und die Parameter einzusetzen. Dieses wird von folgender Regel geleistet, die als Regel (6) schon in der ersten Hälfte aufgetreten ist:

$$[\![(\backslash v_1 ... v_m \to tf)\ t_1 ... t_n]\!] \Rightarrow [\![tf[t_i/v_i]_{i \in 1..m}\ t_{m+1} ... t_n]\!] \qquad (8)$$

Diese Regel wurde stillschweigend schon im Beispiel des vorigen Abschnittes angewendet. Wie oben wird diese Regel auch hier nur angewendet, wenn die Parameter der Lambda-Anwendung nur einmal im Rumpf vorkommen. Ansonsten dient der folgende Schritt der Einführung einer neuen Funktion für die Lambda-Anwendung (a wird wie oben bei Funktionen verwendet):

$$[\![(\backslash v_1 ... v_m \to tf)\ t_1 ... t_n]\!] \Rightarrow f_{\text{NEW}} [\![t_1]\!] ... [\![t_a]\!]\ \vec{w} \tag{9}$$

Hier wird folgende Funktion ($\vec{w}$ sind die Variablen, die im Körper von f_{NEW} nach der Ersetzung von tf noch vorkommen) neu eingeführt:

$$f_{\text{NEW}}\ v_1 \ldots v_a\ \vec{w} = [\![[\![tf]\!]\ [[\![t_i]\!]/v_i]_{i \in a+1..m}\ [\![t_{m+1}]\!] \ldots [\![t_n]\!]]\!]$$

Man beachte die doppelte Anwendung des Verfahrens auf den Rumpf und die Parameter. Es ist bisher ungeklärt, ob die inneren Klammern $[\![]\!]$ wirklich notwendig sind, aber sie vereinfachen den Korrektheitsbeweis.

5.4 Regeln

Hier noch einmal die Regeln, die im obigen zweiten Teil des Verfahrens verwendet werden, im Überblick. Die komplette Beschreibung mit den Vorbedingungen und Seiteneffekten findet sich in den vorangegangenen Abschnitten.

$$\begin{aligned}
[\![v\ t_1 ... t_n]\!] &\Rightarrow v\ [\![t_1]\!] ... [\![t_n]\!] && (1)\\
[\![c\ t_1 ... t_n]\!] &\Rightarrow c\ [\![t_1]\!] ... [\![t_n]\!] && (2)\\
[\![\text{case } t \text{ of } \{p_i \to tf_i\}_{i \in 1..n}]\!] &\Rightarrow \text{case } [\![t]\!] \text{ of } \{p_i \to [\![tf_i]\!]\}_{i \in 1..n} && (3)\\
[\![\backslash v_1 ... v_n \to t]\!] &\Rightarrow \backslash v_1 ... v_n \to [\![t]\!] && (4)\\
[\![f\ t_1 ... t_n]\!] &\Rightarrow f\ [\![t_1]\!] ... [\![t_n]\!] && (5)\\
[\![f\ t_1 ... t_n]\!] &\Rightarrow f_{\text{NEW}} [\![t_1]\!] ... [\![t_a]\!]\ \vec{w} && (6)\\
[\![(\backslash v_1 ... v_m \to tf)\ t_1 ... t_n]\!] &\Rightarrow \backslash w_1 ... w_{n'} \to [\![(\backslash v_1 ... v_m \to tf)\ \vec{t}\ \vec{w}]\!] && (7)\\
[\![(\backslash v_1 ... v_m \to tf)\ t_1 ... t_n]\!] &\Rightarrow [\![tf[t_i/v_i]_{i \in 1..m}\ t_{m+1} ... t_n]\!] && (8)\\
[\![(\backslash v_1 ... v_m \to tf)\ t_1 ... t_n]\!] &\Rightarrow f_{\text{NEW}} [\![t_1]\!] ... [\![t_a]\!]\ \vec{w} && (9)
\end{aligned}$$

6 Fazit

6.1 Grenzen des vorgestellten Verfahrens

Das vorgestellte Verfahren ist in der Lage, die meisten Funktionen höherer Ordnung zu eliminieren. Es gibt jedoch einige Ausnahmen:

- Funktionen, die zur Laufzeit konstruiert werden (wie die vorgestellte Array-Implementierung), lassen sich nicht optimieren
- strukturierte Datentypen, die Funktionen enthalten, werden nicht optimiert

- Funktionsparameter, die in einer Rekursion erweitert werden, können nicht optimiert werden

6.2 Nutzen

Das vorgestellte Verfahren ist in der Lage, die meisten Programme derart zu optimieren, dass keine Funktionen höherer Ordnung mehr auftreten. Praxistests haben gezeigt, daß diese Optimierung für die Haskellimplementierungen `hugs` und `ghci` keine Beschleunigung bringen, aber evtl. machen diese Compiler schon ähnliche Optimierungen.

Einen Nutzwert hat dieses Verfahren insbesondere dann, wenn man Funktionen höherer Ordnung als Abstraktion verwendet, diese auf dem Zielsystem aber nicht implementiert werden können (z.B. für Embedded Systems wegen des beschränkten Speichers).

Literatur

[1] Wei-Ngan Chin and John Darlington. A higher-order removal method. *Lisp Symb. Comput.*, 9(4):287–322, 1996.

[2] Michael Georgeff. Transformations and reduction strategies for typed lambda expressions. *ACM Trans. Program. Lang. Syst.*, 6(4):603–631, 1984.

[3] Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

[4] Philip Wadler. Deforestation: transforming programs to eliminate trees. In *Proceedings of the Second European Symposium on Programming*, pages 231–248. North-Holland Publishing Co., 1988.