Eingereicht zur Veröffentlichung im Sonderband „Die besten Informatikdissertationen des Jahres 1996“, 20. Juli 1997.

Projektionsbasierte Striktheitsanalyse

Ralf Hinze

Institut für Informatik III, Universität Bonn, Römerstraße 164, 53117 Bonn

Zusammenfassung Moderne funktionale Programmiersprachen wie Haskell [9] abstrahieren von der Berechnungsreihenfolge: Die Programmiererin kann sich auf die Beschreibung dessen, *was* berechnet werden soll, konzentrieren, ohne die genaue Reihenfolge der Berechnung spezifizieren zu müssen. Für den erhöhten Komfort bei der Programmierung muß natürlich ein Preis gezahlt werden. Soll dieser nicht in erhöhten Programmlaufzeiten bestehen, so muß ein Übersetzer aufwendige Programmanalysen und -optimierungen durchführen. Die Rekonstruktion der Berechnungsreihenfolge ist die ureigene Aufgabe der Striktheitsanalyse: Die in einem funktionalen Programm nur implizit durch die Datenabhängigkeiten gegebene Abarbeitungsreihenfolge wird explizit gemacht. Meine Dissertation [4] beschäftigt sich mit einer speziellen Instanz der Striktheitsanalyse, die insbesondere die Analyse von Funktionen auf Datenstrukturen wie Listen oder Bäumen ermöglicht. Dieser Artikel gibt einen Überblick über die Arbeit, beleuchtet ihren Hintergrund und benennt die wichtigsten Resultate.

1 Einleitung

Die Dissertation vereinigt zwei meiner wissenschaftlichen Leidenschaften: funktionale Programmierung und Übersetzerbau. Grundlegende Kenntnisse aus beiden Gebieten sind für das Verständnis der Arbeit wenn nicht notwendig, so zumindest doch sehr hilfreich. Aus diesem Grund geben wir in den nächsten vier Abschnitten zunächst eine kurze Einführung in die funktionale Programmierung und in die Übersetzung funktionaler Sprachen. Der Aufbau der Abschnitte 6 bis 9 entspricht im wesentlichen dem der Dissertation; ein Überblick wird am Ende von Abschnitt 5 gegeben.

Eine Vorbemerkung ist noch angebracht: Aufgrund des einführenden Charakters des Artikels werden andere Arbeiten und insbesondere Vorarbeiten nicht oder nicht ausreichend gewürdigt; die Leserin sei hierzu auf das einleitende Kapitel der Dissertation [4] verwiesen.

2 Funktionale Programmierung

Funktionalen Programmiersprachen liegt ein einfaches Rechenmodell zugrunde: Ein Problem wird durch einen Ausdruck beschrieben; um die Lösung des Problems zu ermitteln, wird der Ausdruck zu einem Wert ausgerechnet. Beim Wort „Rechnen“ wird man in erster Linie an das Rechnen mit Zahlen denken; Rechnen ist aber keineswegs auf Arithmetik eingeschränkt. Auch mit Noten und

Partituren [5], mit zweidimensionalen Graphiken [2] oder mit interaktiven, multimedialen Animationen [1] kann man rechnen. Wir schauen uns zur Illustration ein etwas weniger interessantes, dafür aber bekannteres Beispiel an: Es gilt, ein Programm zu schreiben, das eine Folge von ganzen Zahlen sortiert. Um das Sortierproblem zu lösen, müssen zwei Dinge in Angriff genommen werden: Wir müssen uns überlegen, wie wir Folgen repräsentieren und wir müssen Rechenregeln für das Sortieren aufstellen. Für die Notation der Programme verwenden wir im folgenden die Sprache Haskell [9].

Ein Datentyp für Folgen respektive Listen ist in den allermeisten funktionalen Sprachen vordefiniert, so auch in Haskell. Die Liste der ersten vier Primzahlen, zum Beispiel, wird durch den Ausdruck $2 : (3 : (5 : (7 : [\,])))$ beschrieben (die Klammern sind redundant). Eine Liste ist somit entweder leer, $[\,]$, oder von der Form $a : x$, wobei a das erste Listenelement ist und x die restliche Liste ohne das erste Element.

Verschiedene Sortiermethoden führen zu unterschiedlichen Rechenregeln. Eine einfache Methode stellt Sortieren durch Einfügen dar.

$$
\begin{array}{ll}
\mathit{isort} & :: [\mathit{Integer}] \to [\mathit{Integer}] \\
\mathit{isort}\ [\,] & = [\,] \\
\mathit{isort}\ (a : x) & = \mathit{insert}\ a\ (\mathit{isort}\ x) \\
\mathit{insert} & :: \mathit{Integer} \to [\mathit{Integer}] \to [\mathit{Integer}] \\
\mathit{insert}\ a\ [\,] & = a : [\,] \\
\mathit{insert}\ a\ (b : x) & = \mathbf{if}\ a \leqslant b\ \mathbf{then}\ a : b : x\ \mathbf{else}\ b : \mathit{insert}\ a\ x
\end{array}
$$

Die Gleichungen, von links nach rechts gelesen, spezifizieren Rechenregeln. Sie müssen wiederholt angewendet werden, um zum Beispiel die Liste $2 : 3 : 1 : [\,]$ zu sortieren.

$$
\begin{array}{ll}
 & \mathit{isort}\ (2 : 3 : 1 : [\,]) \\
\Rightarrow & \mathit{insert}\ 2\ (\mathit{isort}\ (3 : 1 : [\,])) \\
\Rightarrow & \mathit{insert}\ 2\ (\mathit{insert}\ 3\ (\mathit{isort}\ (1 : [\,]))) \\
\Rightarrow & \mathit{insert}\ 2\ (\mathit{insert}\ 3\ (\mathit{insert}\ 1\ (\mathit{isort}\ [\,]))) \\
\Rightarrow & \mathit{insert}\ 2\ (\mathit{insert}\ 3\ (\mathit{insert}\ 1\ [\,])) \\
\Rightarrow & \mathit{insert}\ 2\ (\mathit{insert}\ 3\ (1 : [\,])) \\
\Rightarrow & \mathit{insert}\ 2\ (1 : \mathit{insert}\ 3\ [\,]) \\
\Rightarrow & 1 : \mathit{insert}\ 2\ (\mathit{insert}\ 3\ [\,]) \\
\Rightarrow & 1 : \mathit{insert}\ 2\ (3 : [\,]) \\
\Rightarrow & 1 : 2 : 3 : [\,]
\end{array}
$$

Typisch für Rechnungen ist, daß es in deren Verlauf mehrere Möglichkeiten gibt fortzufahren. In der obigen Rechnung kann der Ausdruck $\mathit{insert}\ 2\ (1 : \mathit{insert}\ 3\ [\,])$ entweder zu $1 : \mathit{insert}\ 2\ (\mathit{insert}\ 3\ [\,])$ oder zu $\mathit{insert}\ 2\ (1 : 3 : [\,])$ vereinfacht werden. Führt man beide Rechnungen weiter, so gelangt man schließlich zum gleichen Ergebnis. Das ist stets der Fall: Eine fundamentale Eigenschaft rein funktionaler Sprachen besagt, daß das Ergebnis einer Rechnung unabhängig von der Wahl der jeweiligen Vereinfachung und somit eindeutig ist. Es bleibt die Frage, ob unterschiedliche Rechenwege auch stets gleich lang sind. Daß dies nicht

der Fall ist, zeigt das folgende Beispiel (mit (==) wird der Test auf Gleichheit bezeichnet).

$$
\begin{array}{ll}
\mathit{member} & :: \mathit{Integer} \to [\mathit{Integer}] \to \mathit{Bool} \\
\mathit{member}\ a\ [\,] & = \mathit{False} \\
\mathit{member}\ a\ (b : x) & = a == b \vee \mathit{member}\ a\ x \\
(\vee) & :: \mathit{Bool} \to \mathit{Bool} \to \mathit{Bool} \\
\mathit{False} \vee b & = b \\
\mathit{True} \vee b & = \mathit{True}
\end{array}
$$

Der Aufruf $\mathit{member}\ a\ x$ überprüft, ob a in der Liste x enthalten ist. Wir erwarten natürlich, daß die Suche beendet wird, sobald a gefunden ist. Schauen wir uns eine Beispielrechnung an.

$$
\begin{array}{ll}
 & \mathit{member}\ 2\ (1 : 2 : 3 : 4 : 5 : 6 : [\,]) \\
\Rightarrow & 1 == 2 \vee \mathit{member}\ 2\ (2 : 3 : 4 : 5 : 6 : [\,]) \\
\Rightarrow & \mathit{False} \vee \mathit{member}\ 2\ (2 : 3 : 4 : 5 : 6 : [\,]) \\
\Rightarrow & \mathit{member}\ 2\ (2 : 3 : 4 : 5 : 6 : [\,]) \\
\Rightarrow & 2 == 2 \vee \mathit{member}\ 2\ (3 : 4 : 5 : 6 : [\,]) \\
\Rightarrow & \mathit{True} \vee \mathit{member}\ 2\ (3 : 4 : 5 : 6 : [\,]) \\
\Rightarrow & \mathit{True}
\end{array}
$$

Das ist die kürzeste Rechnung. Ungeschickt wäre es gewesen, im vorletzten Schritt den Teilausdruck $\mathit{member}\ 2\ (3 : 4 : 5 : 6 : [\,])$ zu vereinfachen. Warum? Nun, der Gesamtausdruck kann mit der zweiten Regel für $(\vee)$ zu *True* ausgewertet werden, *ohne* daß wir den Wert des Teilausdrucks bestimmen müssen.

Tatsächlich läßt sich eine *Strategie* angeben, die unnötige Rechenschritte vermeidet. Im Englischen wird diese Strategie „lazy evaluation“ genannt, zu Deutsch faule Auswertung. Technisch gesehen erfolgt dabei die Auswertung von außen nach innen: In jedem Schritt wird der äußerste linke Teilausdruck, auf den eine Regel anwendbar ist, vereinfacht. Diese Vorgehensweise garantiert, daß nicht unnötig gerechnet wird, denn: Um einen Wert zu erhalten, muß der äußerste linke Funktionsaufruf auf jeden Fall ausgerechnet werden.

So natürlich „lazy evaluation“ auch erscheint, gängiger ist die duale Strategie: die Auswertung von innen nach außen, im Englischen „eager evaluation“. Warum dem so ist, werden wir im nächsten Abschnitt sehen.

Beide Strategien lassen sich auch semantisch charakterisieren: Ein „Semantiker“ würde eine Sprache mit „lazy evaluation“ als nicht-strikt bezeichnen, eine „eager“ Sprache entsprechend als strikt. Aus den semantischen Begriffen leitet sich auch der Terminus Striktheitsanalyse ab.

Interessant ist, daß die Wahl einer bestimmten Auswertungsstrategie großen Einfluß auf die Programmierung hat. In einer strikten Sprache würde man die Funktion *member* anders formulieren: Die vorzeitige Beendung der Suche müßte explizit mit einer Kontrollstruktur programmiert werden. Leider können wir aus Platzgründen nicht auf die Vorteile von „lazy evaluation“ eingehen. Diese werden aber von John Hughes sehr schön in dem Artikel „Why Functional Programming Matters“ [7] dargestellt und mit vielen Beispielen illustriert.

3 Implementierung von „lazy evaluation“

Verglichen mit „eager evaluation“ ist die Implementierung von „lazy evaluation“ wesentlich aufwendiger und führt zu langsameren Programmen – zumindest auf konventionellen „von Neumann“-Architekturen. Warum dem so ist, läßt sich gut mit Hilfe des letzten Beispiels verdeutlichen.

In Sprachen wie Pascal oder C werden stets Werte als Parameter übergeben bzw. Objekte, die sich gut als Maschinenwert repräsentieren lassen; die Funktion $(\vee)$ erhält im Gegensatz dazu einen *unausgewerteten* Ausdruck als zweiten Parameter. Die Übergabe eines Ausdruck ist hier zwingend: Werten wir den Parameter aus, so durchläuft *member* die Eingabeliste stets bis zum Ende.

Die Vorteile von „lazy evaluation“ werden mit erhöhtem Aufwand bei der Übergabe von Parametern – es werden Ausdrücke nicht Werte übergeben – und beim Zugriff auf Parameter – die an die Parameter gebundenen Ausdrücke müssen ausgerechnet werden – erkauft. Eine „eager“ Sprache läßt sich im Gegensatz dazu wesentlich leichter auf eine konventionelle Rechnerarchitektur abbilden.

Interessanterweise kann man eine für „eager evaluation“ konzipierte Maschine für „lazy evaluation“ aufrüsten, indem man einen Datentyp „Berechnungsvorschrift“ einführt. Eine Berechnungsvorschrift (engl. recipe, laze, thunk oder closure) ist sozusagen die Maschinendarstellung eines unausgewerteten Ausdrucks; sie besteht typischerweise aus dem für den Ausdruck generierten Maschinencode und den Belegungen der in dem Ausdruck auftretenden freien Variablen.

Die Rückführung von „lazy“ auf „eager evaluation“ ist insbesondere von Bedeutung, weil sie den mit „lazy evaluation“ verbundenen Aufwand plastisch vor Augen führt. Am einfachsten läßt sich diese Rückführung durch eine Programmtransformation bewerkstelligen. Die Konstruktion einer Berechnungsvorschrift machen wir mit *freeze* explizit, die Ausführung einer Berechnungsvorschrift mit *unfreeze* bzw. durch Angabe von *freeze* auf der linken Seite einer Gleichung. Wenden wir die Transformation auf die Definition von *member* und $(\vee)$ an, erhalten wir das folgende Programm.

$$
\begin{aligned}
&\mathit{member}\ a\ (\mathit{freeze}\ [\,]) &&= \mathit{False}\\
&\mathit{member}\ a\ (\mathit{freeze}\ (b : x)) &&= \mathit{freeze}\ (a == b) \vee \mathit{freeze}\ (\mathit{member}\ a\ x)\\
&(\mathit{freeze}\ \mathit{False}) \vee b &&= \mathit{unfreeze}\ b\\
&(\mathit{freeze}\ \mathit{True}) \vee b &&= \mathit{True}
\end{aligned}
$$

Selbst für dieses vergleichsweise einfache Programm entsteht ein beträchtlicher Mehraufwand. Man beachte, daß neben den Parametern von Funktionen auch die Argumente des Listenkonstruktors „:“ verzögert werden. Aus dem Ausdruck $\mathit{member}\ 2\ (1 : 2 : 3 : 4 : 5 : 6 : [\,])$ wird

$$
\begin{aligned}
&\mathit{member}\ (\mathit{freeze}\ 2)\ (\mathit{freeze}\ (\mathit{freeze}\ 1 : \mathit{freeze}\ (\mathit{freeze}\ 2 : \mathit{freeze}\ (\mathit{freeze}\ 3 :\\
&\quad \mathit{freeze}\ (\mathit{freeze}\ 4 : \mathit{freeze}\ (\mathit{freeze}\ 5 : \mathit{freeze}\ (\mathit{freeze}\ 6 : \mathit{freeze}\ [\,])))))))\ .
\end{aligned}
$$

4 Striktheitsanalyse

Die Aufgabe der Striktheitsanalyse ist es, Funktionsargumente zu identifizieren, die „eager“ übergeben werden können, ohne die Bedeutung des Programms zu verändern. Im obigen Programm kann zum Beispiel der erste Parameter von $(\vee)$ ausgewertet werden; entsprechendes gilt für das zweite Argument von *member*. Die Argumente werden jeweils benötigt, um entscheiden zu können, welche Gleichung zur Anwendung kommt. Basierend auf diesen einfachen Überlegungen läßt sich das transformierte Programm bereits optimieren.

$$
\begin{array}{ll}
\mathit{member}\ a\ [\,] & = \mathit{False} \\
\mathit{member}\ a\ (b : x) & = a == b \vee \mathit{freeze}\ (\mathit{member}\ a\ (\mathit{unfreeze}\ x)) \\
\mathit{False} \vee b & = \mathit{unfreeze}\ b \\
\mathit{True} \vee b & = \mathit{True}
\end{array}
$$

Wir erhalten ein Programm, in dem Argumente sowohl „lazy“ als auch „eager“ übergeben werden. Empirische Untersuchungen zeigen, daß Optimierungen dieser Art Laufzeitverbesserungen von durchschnittlich 10%–20% bewirken [10].

Aber es gibt noch weiteres Optimierungspotential: Bisher haben wir gefragt, ob ein Funktionsargument benötigt wird oder nicht. Wenn die Argumente Datenstrukturen wie Listen oder Bäume sind, kann man einen Schritt weitergehen und auch die Infrastruktur der Argumente untersuchen. Optimierungen sind bei rekursiven Datenstrukturen besonders lohnend, da der Gewinn unter Umständen proportional zur Größe der Datenstruktur ist.

Jetzt wird die Angelegenheit interessant. Kommen wir auf unser erstes Beispiel zurück, Sortieren durch Einfügen. Auf den ersten Blick würde man vermuten, daß alle Elemente der Eingabeliste benötigt werden, um eine sortierte Ausgabeliste zu erzeugen. Das ist fast richtig, aber eben nur fast. Betrachten wir die beiden Aufrufe *length* (*isort* x) und *head* (*isort* x), wobei *length* und *head* wie folgt definiert sind.

$$
\begin{array}{ll}
\mathit{length} & :: [\alpha] \to \mathit{Integer} \\
\mathit{length}\ [\,] & = 0 \\
\mathit{length}\ (a : x) & = 1 + \mathit{length}\ x \\
\mathit{head} & :: [\alpha] \to \alpha \\
\mathit{head}\ (a : x) & = a
\end{array}
$$

Die Funktion *length* bestimmt die Länge einer Liste; *head* extrahiert das erste Element, das Kopfelement, einer Liste. Setzen wir in den obigen Ausdrücken für x jeweils die Liste *undefined* : [ ] ein, so wertet der erste Ausdruck zu 1 und der zweite zu *undefined* aus.[1] Erstaunlicherweise wird im ersten Fall nicht auf das Listenelement zugegriffen – sonst wäre das Ergebnis ebenfalls undefiniert. Die Erklärung fällt nicht schwer: Eine einelementige Liste kann sortiert werden, ohne das Listenelement zu kennen. Ärgerlich ist, daß dieser Sonderfall eine

[1] Der Ausdruck *undefined* steht für einen undefinierten Wert; die Auswertung von *undefined* führt zu einem unmittelbaren Fehlerabbruch.

Optimierung verhindert. Würden wir die Listenelemente vor dem Aufruf von *isort* ausrechnen, würde sich im obigen Beispiel die Bedeutung des Programms ändern: statt 1 wäre das Ergebnis von *length* (*isort* x) dann *undefined*.

Die Beispiele illustrieren einen zweiten Punkt: Wollen wir das Verhalten von Funktionen adäquat charakterisieren, müssen wir den Kontext berücksichtigen, in dem ein Funktionsaufruf steht. Die Funktion *isort* benötigt die Elemente der Eingabeliste nur, wenn auch die Elemente der Ergebnisliste benötigt werden. Liegt dieser Fall vor, dann kann die oben beschriebene Optimierung durchgeführt werden.

5 Abstrakte Interpretation

Nachdem wir motiviert haben, welche Programmeigenschaften in Hinblick auf die ins Auge gefaßte Optimierung interessant und relevant sind, gilt es zu überlegen, wie diese Eigenschaften automatisch hergeleitet werden können. Leider stoßen wir unmittelbar an die Grenzen der Mechanisierbarkeit: Die Striktheit einer Funktion ist formal unentscheidbar; man kann kein Programm angeben, das feststellt, ob eine Funktion in einem gegebenen Argument strikt ist oder nicht.

Praktisch bedeutet dies, daß ein Analysewerkzeug stets nur Approximationen des tatsächlichen Striktheitsverhaltens ermitteln kann. Nun ist gegen Approximationen nichts einzuwenden, solange sie „von der richtigen Seite“ approximieren: Ist eine Funktion strikt, so darf das Analyseergebnis „nicht strikt“ lauten – in diesem Fall wird Optimierungspotential verschenkt. Der umgekehrte Fall, eine nicht-strikte Funktion wird als strikt klassifiziert, ist hingegen nicht zulässig – die auf Grundlage dieser Information erfolgende Optimierung würde die Bedeutung des Programms verändern.

Den formalen Rahmen für die approximative Programmanalyse bildet die „Abstrakte Interpretation“. Der Name deute bereits die prinzipielle Idee an: Es wird nicht mit konkreten, sondern mit abstrakten Werten gerechnet, die gerade die Eigenschaften modellieren, für die man sich interessiert.

Dem ersten Beispiel für abstrakte Interpretation begegnet man bereits in der Grundschule: Die sogenannte Vorzeichenregel für die Multiplikation erklärt, wie sich das Vorzeichen des Produkts aus den Vorzeichen der beiden Faktoren ergibt, z. B. Minus mal Minus ist Plus. Mit Hilfe der Vorzeichenregel können wir schnell das Vorzeichen des Produkts $-1791 * 65113$ bestimmen, ohne tatsächlich eine Multiplikation durchführen zu müssen. Das gleiche Ziel verfolgen wir auch bei der Striktheitsanalyse: Wir wollen Eigenschaften herleiten, ohne das zu analysierende Programm tatsächlich ausführen zu müssen.

Die „Zutaten“ der Abstrakten Interpretation sind in Abb. 1 dargestellt. Vorgegeben ist die Syntax und die Semantik der Programmiersprache. An ihnen muß sich die Analyse und auch der Nachweis ihrer Korrektheit orientieren. Die Eigenschaften, an denen man interessiert ist, werden zunächst semantisch modelliert, das Ergebnis ist die sogenannte Nichtstandardsemantik. Bei der Modellierung kümmert man sich nicht um Fragen der Berechenbarkeit oder gar

Effizienz. Diesen Aspekten wird erst in einem zweiten Schritt Rechnung getragen, bei der Aufstellung der Approximationssemantik. Diese stellt sozusagen die konstruktive, d.h. effektiv durchführbare Variante der Nichtstandardsemantik dar. Entsprechend gliedert sich auch der Korrektheitsbeweis in zwei Teile: dem Nachweis, daß die Modellierung korrekt ist, und dem Nachweis, daß die Approximationssemantik „von der richtigen Seite“ approximiert.

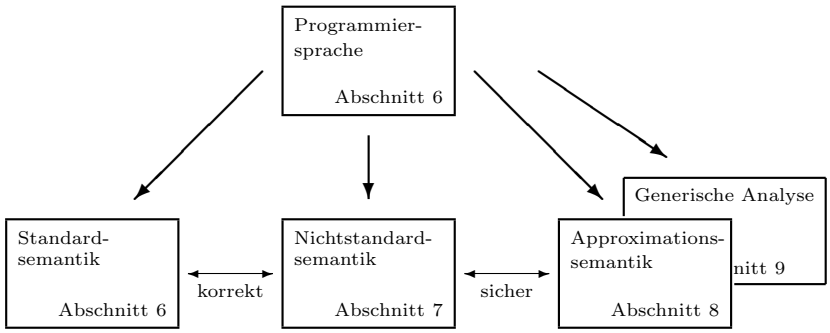

Abbildung1. Abstrakte Interpretation

Da sich die projektionsbasierte Striktheitsanalyse gut in den Rahmen der Abstrakten Interpretation einpaßt, stellt Abb. 1 gleichzeitig auch die Gliederung der Dissertation dar. Die nachfolgenden Abschnitte sind analog organisiert; sie geben jeweils einen kurzen Überblick über das bzw. über die entsprechenden Kapitel der Dissertation.

6 Syntax und Standardsemantik

Für die Notation der Beispielprogramme haben wir die Sprache Haskell verwendet. Als Ausgangspunkt für eine Analyse ist Haskell eher ungeeignet, da die Sprache zu viele redundante Konstrukte enthält: Die Funktion *isort* kann in Haskell mindestens auf fünf verschiedene Weisen programmiert werden. Typischerweise wird im funktionalen Übersetzerbau für die Programmanalyse und -optimierung eine auf die notwendigsten Konstrukte reduzierte Kernsprache verwendet.

Auf einer solchen Sprache basiert auch das entwickelte Analysewerkzeug. Interessanterweise läßt sich die Analyse am elegantesten formulieren, wenn eine *strikte* Sprache mit *freeze*- und *unfreeze*-Konstrukten zugrundegelegt wird. Auf diese Weise wird auch die Analyse hybrider Sprachen möglich, die sowohl „lazy“ als auch „eager evaluation“ unterstützen.

Die einzige Einschränkung gegenüber Haskell ist, daß Funktionen höherer Ordnung *nicht* unterstützt werden (siehe Abschnitt 10).

Sowie die Konstrukte der Kernsprache im wesentlichen vorgegeben sind, so ist auch deren Standardsemantik festgelegt. Die in der Arbeit aufgeführte de-

notationelle Semantik dient im wesentlichen als Bezugspunkt für die späteren Korrektheitsbeweise.

7 Projektionen und Nichtstandardsemantik

Halten wir uns noch einmal das Ziel der Optimierung vor Augen: Der aktuelle Parameter eines Funktionsaufrufs $f(e)$ soll soweit wie möglich *vor* dem Aufruf ausgerechnet werden. In Abschnitt 4 haben wir gesehen, daß ein Ausdruck wie $f(e)$ nicht losgelöst vom Kontext betrachten werden kann – die Auswertung eines Ausdrucks bzw. der Grad seiner Auswertung wird durch den Kontext bestimmt, in den der Ausdruck eingebettet ist. Das ist eine Konsequenz aus der Auswertungsstrategie „lazy evaluation“, die die Auswertung von außen nach innen vorschreibt.

Nun müssen wir aufpassen, daß wir bei der Modellierung nicht über das Ziel hinausschießen: Wir wollen nicht alle Kontexte unterscheiden; uns interessiert nur die vom Kontext bewirkte Auswertung und nicht wie der Kontext den Ausdruck weiterverarbeitet. In diesem Sinne sind die Kontexte $head\ (\cdot) + 1$ und $head\ (\cdot) * 2$ gleichwertig, da beide die Auswertung des Listenkopfs bewirken. Aus diesem Grund schränken wir uns auf die Betrachtung von „Auswertern“ ein. Zwei Eigenschaften sind charakteristisch für Auswerter:

1. Beim Rechnen wird Gleiches durch Gleiches ersetzt: Ein Auswerter verändert die Bedeutung eines Ausdrucks nicht. Aber: Die Auswertung eines Ausdrucks kann divergieren.
2. Ein Auswerter wertet „in einem Durchgang“ aus; die zweifache Anwendung eines Auswerters bewirkt keine zusätzliche Auswertung.

Funktionen mit diesen Eigenschaften werden in der Bereichstheorie *Projektionen* genannt ($\pi\colon D \to D$ ist eine Projektion gdw. $\pi \sqsubseteq \mathrm{id}$ und $\pi \circ \pi = \pi$). Projektionen sind keineswegs für die Striktheitsanalyse erfunden worden. Im Gegenteil: Es dauerte einige Zeit, bis man erkannte, daß sie in idealer Weise für die Modellierung von Auswertern geeignet sind.[2] Projektionen lassen sich gemäß ihres Grades an Auswertung anordnen; je mehr ausgewertet wird, desto kleiner ist die Projektion. Die größte Projektion ist die Identitätsfunktion; sie modelliert den „faulsten“ Auswerter.

Das Auswertungsverhalten einer Funktion kann durch einen *Projektionstransformer* beschrieben werden: Ein Projektionstransformer bildet Projektionen, die die gewünschte Auswertung des Funktionsergebnisses beschreiben, auf Projektionen ab, die die dafür notwendige Auswertung der Funktionsargumente charakterisieren. Ist die vom Projektionstransformer spezifizierte Auswertung „auf der sicheren Seite“, so nennen wir ihn kurz *Abstraktion* der Funktion.

Eine Funktion besitzt viele Abstraktionen; auch hier gilt: je kleiner die Abstraktion ist, desto informativer und besser ist sie. Jede Funktion hat eine größte

[2] Die Idee der kontextbasierten Striktheitsanalyse geht auf John Hughes zurück [6]; erst zwei Jahre später erkannte Philip Wadler die Eignung von Projektionen für die Formalisierung der Analyse [11].

Abstraktion; diese sagt „Auswertung ist nie erforderlich“ ($\tau(\pi) = \mathrm{id}$) und macht Optimierungen unmöglich.

Das führt uns zu der nicht-trivialen Frage, ob es zu einer Funktion stets auch eine kleinste Abstraktion existiert. Diese Frage konnte in der Dissertation positiv beantwortet werden. Interessanterweise ist der Übergang von einer Funktion zu ihrer kleinsten Abstraktion nicht mit einem Informationsverlust verbunden: Die kleinste Abstraktion bestimmt eindeutig die Funktion. In anderen Worten: Aus dem „Analyseergebnis“ läßt sich rekonstruieren, welche Funktion analysiert wurde.

Nun kann man sich in diesem Zusammenhang nicht mit Existenzaussagen zufrieden geben; mindestens so wichtig, wenn nicht wichtiger ist die Frage, ob sich die kleinste Abstraktion auch konstruktiv charakterisieren läßt. Genauer formuliert: Kann man für die funktionale Kernsprache eine kompositionale Semantik angeben, die jedem Sprachkonstrukt ihre kleinste Abstraktion zuordnet?

Man kann zeigen, daß dies im allgemeinen nicht möglich ist. Probleme bereiten Funktionen, die inhärent parallel sind, d.h. für deren Abarbeitung Parallelität erforderlich ist. Prominentes Beispiel für eine solche Funktion ist das „parallele Oder“. Im Gegensatz zum „sequentiellen Oder“, das wir in Abschnitt 2 definiert haben, ist das Ergebnis *True*, wenn ein beliebiges Argument zu *True* auswertet. Parallele Funktionen verletzen das Prinzip der Kompositionalität: die kleinste Abstraktion der Komposition $\varphi \circ \psi$ kann *nicht* durch Komposition der kleinsten Abstraktionen von ψ und φ ermittelt werden.

Das Problem tritt nicht auf, wenn man sich bei der Analyse auf sequentielle Funktionen beschränkt. Für diese Funktionsklasse läßt sich systematisch eine kompositionale Nichtstandardsemantik entwickeln. Die Nichtstandardsemantik wird ähnlich wie die Semantik durch eine Menge von Gleichungen angegeben; jede Gleichung spezifiziert, wie zu einem Konstrukt der Kernsprache die kleinste Abstraktion ermittelt wird. Kompositional meint dabei, daß die Bedeutung eines Ausdrucks sich jeweils aus der Bedeutung der Teilausdrücke ableitet: Die Bedeutung von **if** e_1 **then** e_2 **else** e_3 ergibt sich durch Kombination der Bedeutungen von e_1, e_2 und e_3. Die Nichtstandardsemantik repräsentiert das *optimale* Analyseverfahren; sie ist Orientierung und Meßlatte für die zu entwickelnde Approximationssemantik.

Eine Bemerkung noch zu der Fokussierung auf sequentielle Funktionen: Für die Praxis bedeutet dies keine allzu große Einschränkung. In Haskell können lediglich sequentielle Funktionen definiert werden; Parallelität ist nur möglich, wenn zusätzlich parallele Primitive vordefiniert werden. Darüber hinaus bedeutet der Ausschluß paralleler Funktionen nicht, daß diese sich einer Analyse entziehen. Die für den sequentiellen Fall entwickelten Regeln können auch für eine parallele Sprache verwendet werden; nur ist das Analyseergebnis eben nicht mehr optimal.

8 Kontexte und Approximationssemantik

Auswerter sprich Projektionen können beliebig kompliziert sein: Zum Beispiel gibt es für jede natürliche Zahl n eine Projektion, die eine Liste genau bis zum

n-ten Element auswertet. In Hinblick auf eine praktische Umsetzung der Striktheitsanalyse ist diese Vielfalt unerwünscht. Soll die getrennte Übersetzung von Modulen unterstützt werden, so muß sich die hergeleitete Striktheitsinformation kompakt repräsentieren lassen. Aus diesem Grund ist man gezwungen, sich bei der Analyse auf eine kleine Zahl relevanter Projektionen einzuschränken. Die Projektionen müssen sich darüber hinaus endlich darstellen lassen; die syntaktische Repräsentation einer Projektion nennen wir *Kontext*.

Auf Listen sind vier Kontexte gebräuchlich: HT, H, T und L, die zu den vier Möglichkeiten korrespondieren den Ausdruck $h : t$ auszuwerten. Der Kontext T, zum Beispiel, wertet den Listenrest nicht aber den Listenkopf aus. Allen Kontexten ist gemeinsam, daß der Listenrest jeweils genauso ausgerechnet wird, wie die gesamte Liste. Kontexte mit dieser Eigenschaft heißen *uniform*. Der Kontext T wertet die Struktur einer Liste, nicht aber deren Elemente aus; er modelliert damit die von der Funktion *length* vorgenommene Auswertung. Nachfolgend ist der Kontexttransformer für die Funktion *isort* aufgeführt.

$$\begin{aligned}
\text{isort}^\flat(\text{HT}) &= \text{HT!}\\
\text{isort}^\flat(\text{H}) &= \text{HT!}\\
\text{isort}^\flat(\text{T}) &= \text{T!}\\
\text{isort}^\flat(\text{L}) &= \text{T!}
\end{aligned}$$

Das Ausrufungszeichen besagt, daß das Argument so weit ausgerechnet werden muß, bis ein Ausdruck der Form $[\,]$ oder $h : t$ vorliegt; der vorangestellte Kontext gibt an, wie mit $h : t$ weiterverfahren wird. Neben dem Ausrufungszeichen gibt es auch ein Fragezeichen (siehe unten); dieses zeigt an, daß nicht bekannt ist, ob das Argument ausgewertet werden muß oder nicht; hier beschreibt der vorangestellte Kontext die weitere Auswertung für den Fall, daß das Argument tatsächlich benötigt wird.

Sind die Kontexte festgelegt, so ergibt sich die Approximationssemantik fast automatisch. Im wesentlichen müssen alle Operationen auf Projektionen, die in der Nichtstandardsemantik verwendet werden, in entsprechende Operationen auf Kontexten übersetzt werden.

Die einzige Schwierigkeit bereitet die Analyse *polymorpher* Funktionen. Der Prototyp einer polymorphen Funktion ist die Konkatenation von Listen.

$$\begin{aligned}
&\mathit{append} && :: [\alpha] \to [\alpha] \to [\alpha]\\
&\mathit{append}\ [\,]\ y && = y\\
&\mathit{append}\ (a : x)\ y && = a : \mathit{append}\ x\ y
\end{aligned}$$

Zwei Listen können aneinandergehängt werden, ohne daß die Elemente der Listen inspiziert werden müssen. Somit ist *append* auf Listen beliebigen Grundtyps anwendbar. Im Hinblick auf die Analyse stellt sich folgendes Problem: Da der Grundtyp, α, nicht bekannt ist, wissen wir auch nicht, bezüglich welcher Kontexte die Analyse durchgeführt werden muß.

Die Lösung des Problems besteht darin, neben konkreten Kontexten auch Kontextvariablen einzuführen, die als Platzhalter für unbekannte Kontexte dienen. Zusätzlich müssen die Kontextoperationen erweitert werden, um auch Kon-

texte mit Platzhaltern verarbeiten zu können. Für *append* wird der folgende Kontexttransformer hergeleitet.

$$\begin{aligned}
\mathit{append}^\flat(\mathrm{HT}\,\gamma) &= \mathrm{HT}\,\gamma!\;\mathrm{HT}\,\gamma! \\
\mathit{append}^\flat(\mathrm{H}\,\gamma) &= \mathrm{H}\,\gamma!\quad \mathrm{H}\,\gamma? \\
\mathit{append}^\flat(\mathrm{T}\,\gamma) &= \mathrm{T}\,\gamma!\quad \mathrm{T}\,\gamma! \\
\mathit{append}^\flat(\mathrm{L}\,\gamma) &= \mathrm{L}\,\gamma!\quad \mathrm{L}\,\gamma?
\end{aligned}$$

Da *append* die Listenelemente von den Eingabelisten in die Ausgabeliste kopiert, propagieren sich die Anforderungen an die Listenelemente, bezeichnet durch die Kontextvariable γ, jeweils vom Ergebnis zu den Argumenten.

9 Generische Striktheitsanalyse

Trotz der Einschränkung auf uniforme Kontexte bereitet die Analyse von „real world"-Programmen Probleme: Die Laufzeit der Analyse hängt im wesentlichen von der Anzahl der zu untersuchenden Kontexte ab und diese wächst exponentiell zur Größe des jeweiligen Ergebnistyps. Für Datenstrukturen, die etwa in einem in Haskell geschriebenen Übersetzer für Haskell vorkommen, liegt die Anzahl sehr schnell im Bereich von Billionen (oder mehr).

Eine Möglichkeit, der kombinatorischen Explosion zu entkommen, besteht darin, den Kontexttransformer nicht vollständig, sondern bedarfsgetrieben auszurechnen [8]. Den Bedarf geben die in einem Programm tatsächlich auftretenden Aufrufkontexte an. Dieses Vorgehen verträgt sich aber nicht mit der getrennten Übersetzung von Modulen. Zudem wird – wie bereits angesprochen – als Ergebnis der Übersetzung eine kompakte Repräsentation des vollständigen Kontexttransformers gewünscht.

Schauen wir uns zunächst an, wie sich das letztgenannte Problem lösen läßt. Die grundlegende Idee ist einfach: Die Striktheitsinformationen – oben symbolisiert durch „!" und „?" – werden durch Boole'sche Werte codiert; die Kontexttransformer entsprechend durch Boole'sche Funktionen. Die vier Listenkontexte können durch ein Paar Boole'scher Werte (a, b) repräsentiert werden: a gibt an, ob der Listenkopf ausgerechnet wird, b codiert entsprechend die Auswertung des Listenrests. Im Fall von $\mathit{isort}^\flat$ erhalten wir somit eine Boole'sche Funktion des Typs $\mathbb{B}^2 \to \mathbb{B}^3$, für $\mathit{append}^\flat$ ergibt sich eine Funktion des Typs $\mathbb{B}^2 \to \mathbb{B}^6$.[3]

$$\begin{aligned}
\mathrm{isort}^\flat(a, b) &= (a, 0)0 \\
\mathrm{append}^\flat(a, b) &= (a, b)0\ (a, b)b
\end{aligned}$$

Da die Boole'schen Ausdrücke in der Regel sehr klein sind – in den beiden Beispielen treten nur Projektionen und konstante Funktionen auf – ist die resultierende Darstellung tatsächlich sehr kompakt. Die Repräsentation nutzt aus – und das ist der wesentliche Punkt –, daß „reale" Programme kein chaotisches, sondern im Gegenteil ein sehr uniformes Striktheitsverhalten aufweisen.

[3] Der Kontexttransformer für *append* ist tatsächlich etwas komplizierter, da *append* polymorph ist.

Die grundlegende Idee der generischen Striktheitsanalyse ist es nun, auch die Herleitung von Striktheitsinformationen auf Grundlage dieser Darstellung durchzuführen. Dazu müssen die Kontextoperationen durch Operationen auf Boole'schen Ausdrücken simuliert werden; besondere Schwierigkeiten bereiten dabei polymorphe Funktionen. In der Dissertation wird gezeigt, daß dies tatsächlich ohne Informationsverlust möglich ist. Auf diese Weise erhält man eine Reduktion des Analyseproblems auf das Problem der Lösung monotoner Boole'scher Gleichungssysteme. Praktische Erfahrungen zeigen, daß diese Reduktion die Laufzeit des Analyseverfahrens um mehrere Größenordnungen verbessert.

10 Resümee und Ausblick

Gefragt nach den wesentlichen Beiträgen der Dissertation würde ich zwei Ergebnisse nennen: Die Erarbeitung einer soliden theoretischen Basis für die projektionsbasierte Striktheitsanalyse und die Entwicklung eines praktikablen Werkzeugs für die Durchführung der Analyse.

Zwei wichtige Punkte sind noch ungelöst: Die Analyse von Funktionen höherer Ordnung und die Verwendung der Analyseergebnisse für die Codeerzeugung. Der erste Punkt ist von Bedeutung, weil Funktionen höherer Ordnung gerne und oft bei der Programmierung verwendet werden. Funktionen höherer Ordnung werden zudem bei der Übersetzung von Haskell in die Kernsprache eingeführt: Eine überladene Haskell Funktion wird zu einer Funktion höherer Ordnung in der Kernsprache [3]. Beispiele für überladene Funktionen haben wir bereits kennengelernt: Wenn wir den Typ nicht wie in Abschnitt 2 geschehen auf *Integer* einschränken, sind *isort*, *insert* und *member* überladen.

Es gibt nur wenige Arbeiten, die sich mit der Verwendung der Analyseergebnisse bei der Codeerzeugung beschäftigen. Verschiedene Rahmenbedingungen müssen beachtet werden: Die Platzkomplexität der Programme darf sich nicht verschlechtern. Die Größe des erzeugten Codes muß in einem akzeptablen Rahmen bleiben und die Codeerzeugung sollte sich für die getrennte Übersetzung von Modulen eignen. Darüber hinaus gilt es natürlich, der Qualität der hergeleiteten Striktheitsinformationen gerecht zu werden.

Danksagung

Besonderer Dank gilt meinem Doktorvater, Prof. Armin B. Cremers, für seine fortwährende Unterstützung und Ermutigung.

Literatur

1. Elliott, C. und P. Hudak: *Functional Reactive Animation. Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming.*
2. Finne, S. und S. Peyton Jones: *Pictures: A simple structured graphics model.* In: Turner, D. (Hrsg.): *Proceedings of the 1995 Glasgow Workshop on Functional Programming, Ullapool, Scotland*, Workshops in Computing, Springer-Verlag, Juli 1995.

3. Hall, C. V., K. Hammond, S. L. Peyton Jones und P. L. Wadler: *Type classes in Haskell*. ACM Transactions on Programming Languages and Systems, 18(2):109–138, März 1996.
4. Hinze, R.: *Projection-based Strictness Analysis — Theoretical and Practical Aspects*. Inauguraldissertation, Universität Bonn, November 1995.
5. Hudak, P., T. Makucevich, S. Gadde und B. Whong: *Haskore music notation — An algebra of music*. Journal of Functional Programming, 6(3):465–483, Mai 1996.
6. Hughes, J.: *Strictness detection in non-flat domains*. In: Ganzinger, H. und N. D. Jones (Hrsg.): *Proceedings of the Workshop on Programs as Data Objects, Copenhagen, Denmark*, Nr. 217 in *Lecture Notes in Computer Science*, S. 112–135, Berlin, Oktober 1985. Springer-Verlag.
7. Hughes, J.: *Why Functional Programming Matters*. In: Turner, D. A. (Hrsg.): *Research Topics in Functional Programming*, Kap. 2, S. 17–42. Addison-Wesley Publishing Company, Reading, Massachusetts, 1990.
8. Kubiak, R., J. Hughes und J. Launchbury: *Implementing Projection-based Strictness Analysis*. In: Heldal, R., C. Kehler Holst und P. Wadler (Hrsg.): *Functional Programming, Glasgow 1991: Proceedings of the 1991 Workshop, Portree, Isle of Skye*, Workshops in Computing, London, August 1992. Springer-Verlag.
9. Peterson, J. und K. Hammond: *Report on the Programming Language Haskell 1.4, A Non-strict, Purely Functional Language*. Research Report 1106, Yale University, Department of Computer Science, März 1997.
10. Peyton Jones, S. und W. Partain: *Measuring the effectiveness of a simple strictness analyser*. In: O'Donnell, J. T. und K. Hammond (Hrsg.): *Functional Programming, Glasgow 1993: Proceedings of the 1993 Workshop, Ayr, Scotland*, Workshops in Computing, S. 201–221, London, Juli 1994. Springer-Verlag.
11. Wadler, P. und R. Hughes: *Projections for Strictness Analysis*. In: Kahn, G. (Hrsg.): *Functional Programming Languages and Computer Architecture, Portland, Oregon, USA*, Nr. 274 in *Lecture Notes in Computer Science*, S. 385–407, Berlin, September 1987. Springer-Verlag.