Ausschreibung eines Fortgeschrittenenpraktikum

Effiziente Datenstrukturen für Lambda-Bäume

Gert Smolka, Februar 2002

Motivation

Für die Programmiersprache Alice müssen zur Laufzeit Typen dargestellt werden. Die zurzeit implementierte Darstellung benötigt zuviel Speicher und die erforderlichen Operationen laufen zu langsam. Der Schwachpunkt der jetzigen Darstellung scheint zu sein, dass Typen im Speicher mehrfach dargestellt werden. Das dem Praktikum übergeordnete Ziel ist die Entwicklung und Implementierung einer effizienten Datenstruktur für Typen, die Mehrfachdarstellungen vermeidet.

Typdarstellungen

Typen lassen sich durch Ausdrücke des Lambda-Kalküls beschreiben. Hier ist ein Beispiel:

$$\text{unit} + \text{bool} \times \text{int}$$

$$+\ \text{unit}\ (\times\ \text{bool}\ \text{int})$$

$$(+\ \text{unit})\ ((\times\ \text{bool})\ \text{int})$$

Alle drei Zeilen beschreiben denselben Ausdruck und folglich denselben Typ. Die letzte Zeile verzichtet auf notationale Abkürzungen. Den durch die obigen Ausdrücke beschriebenen Typ fassen wir als endlichen Binärbaum auf.

Für die Beschreibung rekursiver Typen verwenden wir die μ-Notation. Zum Beispiel:

$$\mu\ x.\ \text{unit} + x$$

$$\mu\ x.\ \text{unit} + \text{int} \times x$$

Die durch diese Ausdrücke beschriebenen Typen fassen wir als reguläre Binärbäume auf. Statt von regulären Bäumen spricht man auch von rationalen Bäumen.

Ausdrücke, die Teilausdrücke der Form $\mu x.y$ enthalten, verbieten wir. (Sonst müssen wir sagen, welchen Baum $(\mu x.x)\ (\mu x.x)$ beschreibt).

Für die Beschreibung parametrisierter Typen verwenden wir Lambda-Abstraktion:

$$\lambda x.\ \lambda y.\ \lambda z.\ (x\ z)\ (y\ z)$$

Den durch diesen Ausdruck beschriebenen Typ fassen wir als einen endlichen *Lambda-Baum* auf, der über drei Lambda- und vier Variablen-Knoten verfügt, wobei die Variablen-Knoten mit Offsets auf die jeweils bindenden Lambda-Knoten

referieren (eine Variante der de Bruijnsche Notation). Textuell läßt sich dieser Lambda-Baum wie folgt beschreiben:

$$\lambda\ (\lambda\ (\lambda\ (\ (\underline{5}\ \underline{3})\ (\underline{4}\ \underline{3})\ )))$$

Für die Beschreibung parametrisierter rekursiver Typen benötigen wir sowohl λ als auch μ. Beispielsweise kann der parametrisierte Typ für Listen

$$\text{list } a = \text{unit} + a \times \text{list } a$$

durch den folgenden Ausdruck beschrieben werden:

$$\lambda\ a.\ \mu\ x.\ \text{unit} + a \times x$$

Den dadurch beschriebenen Term fassen wir als einen unendlichen Lambda-Baum auf. Wegen der der Rekursion übergeordneten Lambda-Bindungen ist dieser Lambda-Baum nicht regulär. Bei Verwendung von Standard-de-Bruijn-Notation wäre dieser Baum zwar regulär (da dabei nur die zu überspringenden Lambdas gezählt werden), aber Ausdrücke wie $\lambda\ a.\ \mu\ x.\ \lambda\ b.\ a\ x$ beschreiben immer noch irreguläre Bäume.

Formal benötigen wir also drei Dinge:

1. *Ausdrücke.* Hier können wir die Standarddefinitionen des Lambda-Kalküls verwenden.

2. *Lambda-Bäume.* Diese können wir mit Hilfe von Baumbereichen definieren.

3. Eine *Denotationsfunktion*, die Ausdrücke auf Lambda-Bäume abbildet. Die präzise Formulierung dieser Denotationsfunktion ist für Ausdrücke mit μ nicht offensichtlich.

Der zyklische Lambda-Kalkül von Zena Ariola

Zena Ariola hat mit verschiedenen Kollegen einen Lambdakalkül mit rekursivem Let untersucht, der verschränkt rekursive deklariende Gleichungen erlaubt. Ausdrücke mit μ lassen sich wie folgt in Ausdrücke mit rekursivem Let übersetzen:

$$\mu\ x.\ M \quad ===> \quad \text{letrec } x=M \text{ in } x$$

Viele Aspekte von Ariolas Arbeiten sind relevant für unser Vorhaben. Man findet ihre Arbeiten unter ihrer Homepage. Ein guter Einstieg ist das Papier

Zena M. Ariola and Stefan Blom, Cyclic Lambda Calculi, TACS 1997.

Im Gegensatz zu Ariola interessieren wir uns vorerst nicht für die Beta- oder Eta-Reduktion von Ausdrücken.

Abstrakte Datenstruktur für Lambda-Bäume

Unser Ziel ist die Implementierung der folgenden abstrakten Datenstruktur für Lambda-Bäume:

```
type tree

datatype exp =
    C of string          (* constant    *)
  | A of exp * exp       (* application *)
  | T of tree            (* tree        *)
  | V of string          (* variable    *)
  | M of string * exp    (* mu cycle    *)
  | L of string * exp    (* lambda      *)

exception NotClosed

val enter : exp -> tree  (* NotClosed *)

val equal : tree * tree -> tree

val return : tree -> exp
```

Mit Enter können wir Lambda-Bäume in eine Datenbasis eintragen. Die Lambda-Bäume beschreiben wir durch geschlossene Ausdrücke. Dabei können wir auf bereits eingetragene Bäume zurückgreifen. Mit Equal können wir die Gleichheit von Bäumen testen. Mit Return erhalten wir zu einem Baum einen ihn beschreibenden Ausdruck. Return soll den beschreibenden Ausdruck in möglichst kleinen Schritten zurückgeben (durch Verwendung des T-Konstruktors) wie bei lazy evaluation. Bei unendlichen Bäumen soll ein Ausdruck minimaler Größe geliefert werden.

Die Operationen sollen mit folgender Laufzeiten realisiert werden:

- Enter hat quadratische (?) Laufzeit.
- Equal hat konstante Laufzeit.
- Return hat lineare (?) Laufzeit.

Eine solche Implementierung ist nicht trivial. Wesentliche theoretische Vorarbeit wird durch das Papier

Laurent Mauborgne, An incremental unique representation for regular trees, Nordic Journal of Computing 7 (2000)

geleistet.

Aufgabenstellung für das FoPra

So wie gerade beschrieben ist das FoPra auch für zwei Personen ein sehr anspruchsvolles Projekt. Das liegt vor allen daran, dass Mauborgne das Problem nur für reguläre Bäume ohne Lambda gelöst hat. Die Erweiterung der Algorithmen

von Mauborgne auf Lambda-Bäume sind mindestens eine Diplomarbeit wert, bei sauberer Ausarbeitung mit vollständigen Korrektheitsbeweisen.

Für das FoPra vereinbaren wir die folgende Aufgabenstellung: Implementierung der abstrakten Datenstruktur in Standard ML, mit der Einschränkung, dass Enter nur geschlossene Ausdrücke akzeptiert, die nicht gleichzeitig λ und μ enthalten.

Weiterführende Bemerkungen

Sei der *$\lambda\mu$-Kalkül* mit den obigen Ausdrücken, den Reduktionsregeln für β und η, sowie der folgenden Reduktionsregel für μ definiert:

$$\mu x.M \rightarrow M[\mu x.M / x]$$

Dieser Kalkül ist konfluent, da es sich um ein orthogonal combinatory reduction system handelt [Ariola und Klop, LICS 1994, Abschnitt 8].

Wir betrachten jetzt die einfach getypte Version des $\lambda\mu$-Kalküls, bei der wir μ-Ausdrücke auf nullte Stufe (Basistypen) einschränken. Das bedeutet, dass Rekursion nur auf nullter Stufe möglich ist. Diesen Kalkül bezeichnen wir als *τ–Kalkül*. Er hat die wichtige Eigenschaft, dass μ-Reduktion keine neuen β- und η-Redexe einführt. Mit anderen Worten, μ-Reduktion erhält Normalität bzgl. β und η. Das bedeutet, dass Lambda-Abstraktion und nullstufige Rekursion weitgehend entkoppelt sind. Der τ-Kalkül scheint eine gute Basis für die Beschreibung von Typen zu sein.

Die μ-Reduktion entsprechende Gleichung ist nur eine unvollständige Charakterisierung der für μ intendierten Semantik. Beispielsweise kann die Äquivalenz der folgenden zwei Ausdrücke nicht gezeigt werden:

$$\mu x.\ a\ (b\ x) \qquad\qquad a\ (\mu\, x.b\ (a\ x))$$

Der τ -Kalkül alleine liefert uns also keine vollständige Semantik für Typen.

Geschlossenen Ausdrücken des τ-Kalküls können wir wie folgt eine Denotation zuordnen: normalisiere bzgl β und η (konfluent und terminierend) und nehme dann den entsprechenden Lambda-Baum wie oben beschrieben. Diese Semantik ist zwar nicht schön, scheint aber die richtige Gleichheit zu definieren.

Forschungsfrage: Konstruiere eine elegante denotationale Semantik für den τ-Kalkül, die die gerade definierte Gleichheit liefert.

Die Einschränkung auf nullstufige Rekursion scheint die Voraussetzung für eine entscheidbare Gleichheit zu sein (Klassische Papiere über das Entscheidungsproblem für parametrisierte rekursive Typen im Algol 68 Kontext, Reduktion auf Äquivalenzproblem für nichtdeterministische push down automata).

Related Work

Mehr über reguläre Bäume und rekursive Typen kann man aus dem folgenden, technisch hevorragend geschriebenen Papier lernen:

Dexter Kozen, Jens Palsberg, Michael I. Schwartzbach,
Efficient Recursive Subtyping.
Mathematical Structures in Computer Science 5(1): 113-125 (1995).

Besonders interessant ist die Darstellung von μ–Ausdrücken als Automaten, die die beschriebenen regulären Bäume akzeptieren

Die Constraintsprache *CLLS* von Niehren und Kollegen (hier in Saarbrücken) wurde für die semantische Verabeitung natürlicher Sprache entworfen. Technisch handelt es sich dabei um eine prädikatenlogische Sprache erster Stufe, mit der Lambda-Bäume beschrieben werden können. Das Einstiegspapier für CLLS ist:

Markus Egg, Alexander Koller, Joachim Niehren,
The Constraint Language for Lambda Structures.
Journal of Logic, Language, and Information, 2001.
http://www.ps.uni-sb.de/Papers/abstracts/clls2000.html