Entwicklung eines korrekten Übersetzers

für eine funktionale Programmiersprache im Theorembeweiser Coq

Thomas Strathmann
<thomas.strathmann@informatik.uni-oldenburg.de>

14.01.2011

Gliederung

Motivation

- Einsatz formaler Methoden vergebens, wenn Übersetzer fehlerhaft
- Compilerverifikation sehr aufwändig
  - *Beispiel:* Projekt CompCert enthält über 42.000 Zeilen Coq (14% Compiler, 10% Semantik, 76% Beweis)
- formale Verifikation von Übersetzern mithilfe von Theorembeweisern aktives Forschungsfeld
  - Repräsentation abstrakter Syntax mit Variablenbindungen
  - mechanisierte formale Semantik
  - Formalisierung von Programmtransformationen
  - Beweisautomation

Ziele der Arbeit

- Einführung in Theorie funktionaler Programmiersprachen
  - Lambda-Kalkül
  - formale Semantik
  - Übersetzung
- Fallstudie für die Anwendung eines interaktiven, typtheoretischen Beweisassistenten (Coq)
- Überblick aktueller Ansätze zur computergestützten Formalisierung von Programmiersprachen, formaler Semantik, Programmtransformationen

Coq

- interaktiver, taktischer, typtheoretischer Theorembeweiser
- Theorie: *Kalkül der induktiven Konstruktionen*
  - konstruktive Typtheorie + induktive Definitionen
  - abhängig getypter Lambda-Kalkül
  - Beweiser und funktionale Programmiersprache zugleich
  - Programmextraktion

Einfach Getypter Lambda-Kalkül $\lambda_{\to}$

- Formalismus zur Beschreibung berechenbarer Funktionen
- Grundlage vieler funktionaler Sprachen

Menge $\mathcal{T}$ der Typen

- $\mathcal{T}$ enthält *Basistypen* wie nat oder bool
- *Funktionstypen*: Wenn τ_1 und $\tau_2 \in \mathcal{T}$, so ist $(\tau_1 \to \tau_2) \in \mathcal{T}$

Typurteil $M : \tau$ weist Term M den Typ τ zu

Menge Λ der Terme

- Konstanten und Variablen
- Funktionsanwendung: $(F\ G)$
- λ-Abstraktion: $\lambda x : \tau_1 . M : \tau_2$

α-Konversion

- α-Konversion benennt gebundene Variablen um:
  $\lambda x \,.\, M \overset{\alpha}{\leftrightarrow} \lambda y \,.\, M[y/x]$ (wenn y nicht frei in M)
- gebundene Variablen sind nur „Platzhalter“
- α-konvertierbare Terme werden als äquivalent angesehen
- Semantik von $\lambda_{\rightarrow}$-Termen auf α-Äquivalenzklassen definiert
  $\rightsquigarrow$ Identifikation α-kongruenter Terme notwendig für Beweise semantischer Eigenschaften

β-Reduktion und Substitution

- β-Reduktion ist Funktionsanwendung:
  $(\lambda x\,.\,M)\ N \overset{\beta}{\leftrightarrow} M[N/x]$
- Definition der Substitution?

„Naive“ Substitution — **falsch**

$$\begin{aligned} & (\lambda y\,.\,x)[y/x] \\ = \quad & \lambda y\,.\,y \end{aligned}$$

kollisionsfreie Substitution — **richtig**

$$\begin{aligned} & (\lambda y\,.\,x)[y/x] \\ = \quad & (\lambda z\,.\,x[z/y])[y/x] \\ = \quad & \lambda z\,.\,y \end{aligned}$$

Kollisionsfreie Substitution

Substitutionsregeln

$$
\begin{aligned}
x[M/x] &= M \\
(E\ F)[M/x] &= E[M/x]\ F[M/x] \\
(\lambda x\,.\,E)[M/x] &= \lambda x\,.\,E \\
(\lambda y\,.\,E)[M/x] &= \begin{cases} \lambda y\,.\,E[M/x] & x \text{ nicht frei in } E \vee \text{y nicht frei in } M \\ \lambda z\,.\,(E[z/y])[M/x] & \text{sonst} \end{cases}
\end{aligned}
$$

wobei Variablen x, y, z alle verschieden

- wünschenswert, wenn Formalisierung ohne diese komplizierte Definitions auskäme

Termrepräsentation

- Darstellung der abstrakten Syntax der *Objektsprache* in der *Metasprache*
- wichtige Punkte:
  - Ist die Darstellung *adäquat*?
    keine *exotischen* Terme
    $\Leftrightarrow$ Bijektion zwischen Λ und Instanzen der Termrepräsentation
  - α-Kongruenz
  - Substitution
  - Zugriff auf Funktionsrümpfe
- Zwei Klassen von Ansätzen:
  - Repräsentationen erster Stufe (konkrete Darstellungen)
  - Repräsentationen höherer Stufe

Abstrakte Syntax mit benannten Variablen

- alle Variablen werden durch Namen repräsentiert
- α-konvertierbare Terme sind äquivalent
  - Formalisierung von α-Kongruenz nötig
- Substitution muss formalisiert werden

```
Inductive exp : Set :=
| Var : string → exp
| Abs : string → exp → exp
| App : exp → exp → exp.
```

De-Bruijn-Indizes

- Natürliche Zahlen statt Variablennamen
  - *Beispiel*: $\lambda x\,.\,(\lambda y\,.\,y\;x) \rightsquigarrow \lambda(\lambda 0\;1)$
- α-äquivalente Terme syntaktisch gleich
- Substitution mittels arithmetischer Operationen

Definition ohne Wohlgeformheitsbedingung

```
Inductive exp : Set :=
| Var : nat → exp
| Abs : exp → exp
| App : exp → exp → exp.
```

Adäquate Definition mit integrierter W'bedingung

```
Inductive exp (n : nat) : Set :=
| Var : fin n → exp n
| Abs : exp (S n) → exp n
| App : exp n → exp n → exp n.
```

HOAS

- abstrakte Syntax höherer Stufe (higher-order abstract syntax)
- repräsentiert Bindungskonstrukte der Objektsprache durch Bindungskonstrukte der Metasprache
- Termäquivalenz und Substitution der Metasprache nutzen
- nicht möglich in Coq wegen „negativen Vorkommens“ des ersten *exp* in Konstruktor *Abs*

```
Inductive exp : Set :=
| Abs : (exp → exp) → exp
| App : exp → exp → exp.
```

schwache HOAS

- Lösung des Problems: Führe zusätzliche Menge *var* ein:

```
Parameter var : Set.

Inductive exp : Set :=
| Var : var → exp
| Abs : (var → exp) → exp
| App : exp → exp → exp.
```

- Beachte: *var* ist abstrakt
  - keine Fallunterscheidung nach Variablen möglich
  - ... daher exotische Terme ausgeschlossen
  - allerdings auch kein Zugriff auf Rumpf einer Abstraktion
- Lösungsansätze:
  - *var* konkret machen, z.B. *var := nat*
    exotische Terme möglich, müssen explizit ausgeschlossen werden!
  - **parametrisierte HOAS**

Parametrisierte HOAS

- Idee: Typ der Terme mit Typ der Variablen parametrisieren:

```
Section exp.
  Variable var : Set.

  Inductive exp : Set :=
  | Var : var → exp
  | Abs : (var → exp) → exp
  | App : exp → exp → exp.
End exp.

Definition Exp := ∀ var, exp var.
```

- Zugriff auf Funktionsrümpfe möglich, da durch geeignete Instantiierung von *var* Werte verfügbar sind, die in Funktionen vom Typ $var \to exp$ eingesetzt werden können

Repräsentation getypter Terme

- Wie implementiert man Syntax + Typregeln in einem Beweiser?
- zwei Möglichkeiten:
  - *extrinsische* Darstellung: wie in „mathematischer“ Darstellung eine induktive Definition für Syntax und eine für die Typregeln
  - *intrinsische* Darstellung: Typregeln zusammen mit Syntax in einer induktiven Definition umsetzen
- Vorteil der intrinsischen Darstellung: nur wohlgetypte Terme können konstruiert werden $\rightsquigarrow$ keine Wohlgetyptheitsbedingung in Aussagen über Terme nötig

Vergleich der Darstellungen

Bewertungskriterien für Termrepräsentationen

- α-Kongruenz Objektsprache = syntaktische Gleichheit in Metasprache?
- Realisiert Metasprache Substitution der Objektsprache?
- Darstellung adäquat?
- Möglichkeit, Funktionsrümpfe zu inspizieren, vorhanden?

	α-Kongruenz	Subst.	adäquat	Inspektion
Namen			×	×
De-Bruijn	×		(×)	×
HOAS	×	×		
schwache HOAS	×		×	
PHOAS	×		×	×

Semantik

- formale Semantik notwendig für Korrektheitsbeweise von Programmtransformationen
- Zwei wesentliche Richtungen zu betrachten:
  - denotationale Semantik
  - operationale Semantik
- Fragen
  - Welcher Semantikstil am besten geeignet für Korrektheitsbeweise von Programmtransformationen in einer Turing-vollständigen Sprache?
  - Umsetzbarkeit in Coq

Denotationale Semantik

- denotationale (auch mathematische oder Funktions-) Semantik
- ordnet Termen mathematische Ausdrücke zu
  - interpretiert Terme der Objektsprache als Terme der Metasprache
- viele Beweis-/Berechnungsschritte erledigt die Metasprache
  - Termäquivalenz
  - β-Reduktion
- Problem: Coq erlaubt nur terminierende Berechnungen!
  - Turing-vollständige Objektsprache: Welchen Coq-Term einer rekursiven Funktion zuordnen?

Operationale Semantik

- „mechanische“ Sicht der dynamischen Semantik von Programmiersprachen
- Ausführungsschritte als Ableitungsregeln (relationale Semantik)
  - **big-step (natürliche) Semantik**: Relation zwischen Termen und Werten (Ergebnissen) – Ausführung in einem Schritt
  - **small-step (strukturelle) Semantik**: Relation zwischen Termen – i.d.R. mehrere Schritte bis zum Ergebnis
- nicht-terminierende Berechnungen handhabbar
- in Coq: Relation als induktiv definierter Typ
  - Wie Substitution formalisieren?

Implementierung der Substitution in (P)HOAS

- Substitutionsrelation
  - Relation S, sodass $e_1 \, S \, e_2$, gdw. $e_2 = e_1[M/x]$ für ein $x \in \mathcal{V}_{\text{gebunden}}(e_1)$
- Substitutionsfunktion
  - Coq-Funktion *Subst* (*M* : *Exp*) (*e* : *Exp1*) : *Exp*
    - ⋆ wobei *Exp1* : $\forall$*var*, *var* $\rightarrow$ *exp var*
  - liefert $e[M/x]$ wobei x die in e gebundene Variable ist
- „substitution-free operational semantics“
  - vermeidet explizite Substitution
  - Lambda-Abstraktionen als Closures in Liste speichern

Closure als Coq-Funktion

```
Definition closure := val → exp val
```

- statt Substitution: Closure aus Liste holen und auf Argument anwenden

Behandlung von Nicht-Terminierung

- Nicht-Terminierung: divergierender Term e hat keine Normalform
- unbeschränkte Folge von Ableitungsschritten: $e \to e' \to e'' \to \ldots$
- Ansätze:
  - koinduktive Interpretation der big-step Ableitungsregeln (nicht äquivalent zur small-step Semantik!)
  - induktive big-step Ableitungsregeln + koinduktive Relation für divergierende Terme
  - koinduktive trace-basierte Semantik
  - ...


That's all Folks!