Eine universelle Lambda-Kalkül-Programmiersprache und ihr Interpreter

E. Fehr

Universität Bonn

0. Zusammenfassung: Wir erweitern den λ -Kalkül von A.Church zu einer universellen Programmiersprache (λ_p -Kalkül) über einer beliebigen algorithmischen Basis. In Absatz 2 definieren wir in VDL den Lambda-Programm-Interpreter LPI, der alle im λ_p -Kalkül definierbaren Funktionen über dem Konstantenbereich der algorithmischen Basis berechnet. Der LPI ist ein 'fixed-program (reentrant)''-Interpreter mit einer Call-by-Name-Auswertungsstrategie. Auf Grund einer sorgfältigen Organisation des Symboltabellenzugriffs vermeidet der LPI jegliche Umbenennung von Variablen. Die Korrektheit des LPI wird bewiesen durch Anwendung einer induktiven Beweistechnik auf eine Funktion TRACE, die Zustände des LPI abbildet auf teilausgewertete λ_p -Ausdrücke. Eine ausführliche Fassung dieser Arbeit liegt als Institutsbericht unter gleichem Titel vor (s. /FE/). Dort findet sich insbesondere eine Dokumentation der Implementierung des LPI in SIMULA.

1. Der Lambda-Kalkül als Programmiersprache

Sei $\underline{A} = (C; F, P)$ eine algorithmische Basis, wobei C der Konstantenbereich, F eine endliche Menge von Grundfunktionen über C und P eine endliche Menge von Prädikaten auf C ist. Die Menge $L_{\underline{A}}$ der Lambda-Programme (λ_p -Ausdrücke) definieren wir induktiv:

- i) (Atome) $V \cup A \subset L_{\underline{A}}$, wobei V eine abzählbare Menge von Variablen ist und A aus Symbolen für die Elemente aus $\underline{A}$ besteht.
- ii) (Kombinationen) $L_1, L_2 \in L_{\underline{A}} \Rightarrow (L_1 L_2) \in L_{\underline{A}}$
- iii) (Abstraktionen) $x \in V, L \in L_{\underline{A}} \Rightarrow \lambda x L \in L_{\underline{A}}$
- iv) $L_{\underline{A}}$ wird durch i)-iii) vollständig definiert.

Die Semantik des λ_p -Kalküls wird mathematisch definiert durch die Reduktionsregeln des reinen λ -Kalküls und einer Wertreduktion, welche die Anwendung von Systemfunktionen eines Rechners beschreibt. (Im folgenden bezeichnet $S_A^x M$ die Substitution von A für alle freien Vorkommen von x in M und das Symbol ω steht für undefiniert.)

- i) (α -Konversion, Variablenbenennung) Sei $x, y \in V, L \in L_{\underline{A}}$ und y nicht frei in L, dann gilt: $\lambda x L \rightarrow \lambda y S_y^x L$
- ii) (β -Reduktion) Sei $x \in V, M, A \in L_{\underline{A}}$ und keine Variable kommt frei in A und gebunden in M vor, dann gilt:

$$u(\lambda x M A) v \longrightarrow u S_A^{X M} v$$

iii) (Wertreduktion) Sei $h \in F \cup P$ n -stellig, $a_i \in C$ für $1 \leq i \leq n$ und sei $h(a_1, \dots, a_n) = a$, dann gilt:

$$u(\dots(h a_1) \dots a_n) v \longrightarrow \begin{cases} u a v & \text{falls } a \in C \\ u \lambda x \lambda y x v & \text{falls } a = \underline{\text{true}} \\ u \lambda x \lambda y y v & \text{falls } a = \underline{\text{false}} \\ \omega & \text{falls } a = \omega \end{cases}$$

Ein λ_P -Ausdruck L_2 ist Normalform eines λ_P -Ausdrucks L_1 , wenn $L_1 \xrightarrow{*} L_2$ und keine β -Reduktion und keine Wertreduktion auf L_2 anwendbar ist.

Definition (Semantikfunktion)

$$\sigma : L_A \times C^* \longrightarrow C$$

$$\sigma(L, (c_1, \dots, c_n)) := \begin{cases} c & \text{falls } c \in C \text{ und } c \text{ ist Normalform von} \\ & ((\dots(L c_1) \dots) c_n) \\ \omega & \text{sonst} \end{cases}$$

Sei n die kanonische Einbettung der Programm-Daten-Kombinationen in die Menge L_A der λ_P -Ausdrücke, v die Ergebnisfunktion der Call-by-Name-Berechnung (d.h. Reduktionsfolge stets von links nach rechts) und ψ die Projektion der Menge L_A auf C , dann folgt unmittelbar aus dem Standardization-Theorem von Church-Rosser (s. /CF/ Seite 139 ff), daß folgendes Diagramm kommutiert:

$$\begin{array}{ccc} L_A \times C^* & \xrightarrow{n} & L_A \\ \sigma \downarrow & & \downarrow v \\ C & \xleftarrow{\psi} & L_A \end{array}$$

Bemerkung: Die Menge der im λ_P -Kalkül berechenbaren Funktionen wird nicht eingeschränkt, wenn man nur Programme zuläßt, in denen keine freien Variablen vorkommen (s. /FE/, Lemma II.4.1).

Sei im folgenden $\text{Fr}(L) := \{x \in V \mid x \text{ kommt frei in } L \text{ vor}\}$ und

$$L_A^+ := \{L \in L_A \mid \text{Fr}(L) = \emptyset\}.$$

Lemma: Sei $L_1 \in L_A^+$ und $L_1, L_2, \dots, L_n$ die Call-by-Name-Berechnung von L_1 (d.h. Reduktionsreihenfolge stets von links nach rechts), dann gilt:

$$L_n \in C \Rightarrow \bigvee_{1 \leq i \leq n} (L_i \xrightarrow{\beta} L_{i+1} \text{ oder } L_i \xrightarrow{w} L_{i+1}).$$

Beweis: siehe /FE/.

2. Der Lambda-Programm-Interpreter LPI

Zur Definition der operationellen Semantik benutzen wir die Vienna Definition Language VDL. Eine Einführung in diese Sprache findet man in /WV/.

Der LPI erwartet als Eingabe ein strukturiertes Objekt, das dem VDL-

Prädikat is-expression genügt.

Definition: is-expression := is-atom $\cup$ is-combination $\cup$ is-abstraction

is-atom := $A \cup V$

is-combination := ($\langle s\text{-operator} \rangle : \langle \text{is-expression} \rangle$,
 $\langle s\text{-operand} \rangle : \langle \text{is-expression} \rangle$)

is-abstraction := ($\langle s\text{-boundvar} \rangle : \langle \text{is-var} \rangle$, $\langle s\text{-body} \rangle : \langle \text{is-expression} \rangle$)

is-var := V

Wegen der Eindeutigkeit der Menge L_A existiert eine bijektive Abbildung $\gamma : L_A \rightarrow \text{is-expression}$, so daß die Ergebnisse aus Absatz 1 analog für die Menge is-expression gelten.

Der LPI besteht aus vier Komponenten:

1. Ein Pointer s-work, der jeweils auf das nächste, auszuwertende Programmteil zeigt.
2. Ein Kellerspeicher s-proc (procedure-stack), in den ein Eintrag bei Prozeduraufrufen und Blockaktivierungen gemacht wird.
3. Eine Symboltabelle s-env, auf die Zugriff über einen Environmentvektor erfolgt, dessen Komponenten gemäß der 'static-chain' geordnet sind.
4. Ein Hilfskellerspeicher s-aux, in dem Eintragungen aus s-proc vorübergehend abgespeichert werden.

Das VDL-Prädikat is-state charakterisiert die Zustandsmenge Σ des LPI:

is-state := ($\langle s\text{-work} : \text{is-expression} \rangle$, $\langle s\text{-proc} : \text{is-proc-stack} \rangle$,
 $\langle s\text{-aux} : \text{is-proc-stack} \rangle$, $\langle s\text{-env} : \text{is-environment} \rangle$)

is-proc-stack := ($\langle s\text{-top} : \text{is-proc-entry} \rangle$,
 $\langle s\text{-rest} : \text{is-proc-stack} \rangle$) $\cup$ is-null

is-proc-entry := ($\langle s\text{-control} : \text{is-control-symbol} \rangle$,
 $\langle s\text{-return} : \text{is-return-link} \rangle$, $\langle s\text{-list} : \text{is-number-list} \rangle$)

is-return-link := is-expression $\cup$ is-null

is-null := $\{ \Omega \}$

is-control-symbol := $\{D, C, L, R\}$

is-number-list := ($\langle s\text{-top} : \text{is-number} \rangle$,
 $\langle s\text{-rest} : \text{is-number-list} \rangle$) $\cup$ is-null

is-Number := $\mathbb{N}$

is-table-entry := ($\langle s\text{-var} : \text{is-var} \rangle$, $\langle s\text{-wert} : \text{is-expression} \rangle$,
 $\langle s\text{-list} : \text{is-number-list} \rangle$)

Das Prädikat is-environment wird durch die Angabe seines Domain und Range definiert.

is-environment(e) : $\Leftrightarrow D(e) \subset \{s\text{-}i \mid i \in \mathbb{N}\}$ und
 $R(e) \subset \text{is-table-entry}$

Zur Definition der Zustandsüberföhrungsfunktion Δ des LPI benötigen wir einige Abkürzungen und Hilfsfunktionen:

S-LISTE := s-list • s-top • s-proc (Environmentvektor)

S-CONTROL := s-control • s-top • s-proc

S-RETURN := s-return • s-top • s-proc (Rücksprungadresse)

Die formale Definition in VDL der nun folgenden Funktionen und Prädikate findet man in /FE/ Seite 42 ff, hier nur eine kurze Beschreibung:

Die Funktion δ -value reduziert den am weitesten links stehenden Wert-Redex, falls ein solcher existiert.

Das Prädikat is-equ bezeichnet die Äquivalenz zweier Ausdrücke und das Prädikat i-C-expr wird von Konstantenausdrücken erfüllt.

Die Kellerfunktionen pushdown und popup sind wie üblich definiert, und die Funktion push angewendet auf zwei Keller überträgt den Inhalt des zweiten Kellers in der last-in-first-out-Reihenfolge auf den ersten Keller. Und schließlich die Funktion newnumber, die angewendet auf eine Symboltabelle, die erste, freie Adresse liefert.

Definition: (Die Transitionsfunktion Δ des LPI)

$\Delta : \Sigma \longrightarrow \Sigma$

$\Delta(\beta) := \underline{\text{is-C-expr}}(s\text{-work}(\beta)) \longrightarrow$

$\text{not is-equ}(s\text{-work}(\beta), \delta\text{-value}(s\text{-work}(\beta))) \longrightarrow$

$\mu(\beta; \langle s\text{-work} : \delta\text{-value}(s\text{-work}(\beta)) \rangle)$

$\text{is-C or is-D}(S\text{-CONTROL}(\beta)) \longrightarrow$

$\mu(\beta; \langle s\text{-work} : S\text{-RETURN}(\beta),$

$\langle S\text{-RETURN} : s\text{-work}(\beta),$

$\langle S\text{-CONTROL} : R \rangle)$

$\text{is-R}(S\text{-Control}(\beta)) \longrightarrow$

$\mu(\beta; \langle s\text{-work} : \mu_0(\langle s\text{-operator} : S\text{-RETURN}(\beta),$

$\langle s\text{-operand} : s\text{-work}(\beta) \rangle),$

$\langle s\text{-proc} : \text{popup}(s\text{-proc}(\beta)) \rangle \rangle)$

$\underline{\text{is-abstraction}}(s\text{-work}(\beta)) \longrightarrow$

$\text{is-null}(s\text{-aux}(\beta)) \longrightarrow$

$\mu(\beta; \langle s\text{-top} \cdot s\text{-aux} : \mu_0(\langle s\text{-control} : D, \langle s\text{-list} :$

$\text{pushdown}(S\text{-LISTE}(\beta), \text{newnumber}(\beta)) \rangle \rangle)$

$\text{is-L}(S\text{-CONTROL}(\beta)) \longrightarrow$

$\mu(\beta; \langle s\text{-work} : s\text{-body} \cdot s\text{-work}(\beta),$

$\langle s\text{-newnumber}(\beta) \cdot s\text{-env} : \mu_0(\langle s\text{-var} : s\text{-boundvar}(s\text{-work}(\beta)) \rangle,$

$\langle s\text{-wert} : S\text{-RETURN}(\beta) \rangle, \langle s\text{-list} : S\text{-LISTE}(\beta) \rangle \rangle,$

$\langle s\text{-proc} : \text{push}(\text{popup}(s\text{-proc}(\beta)), s\text{-aux}(\beta)) \rangle, \langle s\text{-aux} : \Omega \rangle)$

$\text{is-C or is-D}(S\text{-CONTROL}(\beta)) \longrightarrow$

$\mu(\beta; \langle s\text{-proc} : \text{popup}(s\text{-proc}(\beta)) \rangle, \langle s\text{-aux} : \text{pushdown}(s\text{-aux}(\beta),$

$\langle s\text{-top} \cdot s\text{-proc}(\beta) \rangle \rangle)$

```

is-combination(s-work( $\beta$ ))  $\longrightarrow$ 
   $\mu(\beta; \langle s\text{-work:s-operator} \cdot s\text{-work}(\beta) \rangle,$ 
     $\langle s\text{-proc:pushdown}(s\text{-proc}(\beta),$ 
       $\mu_0(\langle s\text{-control:L} \rangle, \langle s\text{-return:s-operand} \cdot s\text{-work}(\beta) \rangle,$ 
         $\langle s\text{-list:S-LISTE}(\beta) \rangle \rangle) \rangle)$ 

is-var(s-work( $\beta$ ))  $\longrightarrow$ 
  not is-C(S-CONTROL( $\beta$ ))  $\longrightarrow$ 
     $\mu(\beta; \langle s\text{-proc:pushdown}(s\text{-proc}(\beta),$ 
       $\mu_0(\langle s\text{-control:C} \rangle, \langle s\text{-list:S-LISTE}(\beta) \rangle) \rangle)$ 
is-equ(s-work( $\beta$ ), var(S-LISTE( $\beta$ ),  $\beta$ ))  $\longrightarrow$ 
   $\mu(\beta; \langle s\text{-work:wert}(S\text{-LISTE}(\beta), \beta) \rangle,$ 
     $\langle S\text{-LISTE:list}(S\text{-LISTE}(\beta), \beta) \rangle)$ 
not is-null (S-LISTE( $\beta$ ))  $\longrightarrow$ 
   $\mu(\beta; \langle S\text{-LISTE:popup}(S\text{-LISTE}(\beta)) \rangle)$ 

```

Um die Korrektheit des LPI zu beweisen, definieren wir die Funktion TRACE, die die Zustandsmenge Σ des LPI abbildet auf die Menge L_A^+ der λ_p -Ausdrücke. In der Definition von TRACE benutzen wir die Funktion SUBSTITUTE, die alle freien Variablen eines λ_p -Ausdrucks durch ihre Werte, gemäß dem aktuellen Environment, ersetzt.

Für einen Zustand β aus Σ wird TRACE(β) wie folgt berechnet:

Falls s-proc leer ist, ist TRACE(β) gleich dem Inhalt von s-work, sonst transformieren wir alle Information, die in s-proc enthalten ist sukzessive in einen λ_p -Ausdruck zurück, indem wir jeweils die Umkehrung von Δ ausführen und wenn nötig die Funktion SUBSTITUTE anwenden (Def. in VDL siehe /FE/).

Die Korrektheit des LPI ergibt sich nun aus der Kommutativität des folgenden Diagramms:

wobei $\text{INPUT}(L) := \mu_0(\langle s\text{-work:}\gamma(L) \rangle)$,
 $L_A^+ := \{L/L \in L_A \wedge \text{Fr}(L) = \emptyset\}$ und
 η, v, Ψ wie in Abschnitt 1

Der Beweis besteht aus drei Teilen:

Sei $(P, \bar{c}) \in L_A^+ \times C^*$ und $\xi_{(P, \bar{c})}^0, \xi_{(P, \bar{c})}^1, \dots$ die entsprechende Zustandsfolge des LPI.

Teil 1: $\text{trace}(\xi_{(P, \bar{c})}^0) = n(P, \bar{c})$

Teil 2: $\bigvee_{i \in \mathbb{N}} \xi_{(P, \bar{c})}^i \neq \Omega \implies$

$$\xi_{(P, \bar{c})}^{i+1} = \Omega \text{ oder}$$

$$\text{trace}(\xi_{(P, \bar{c})}^i) = \text{trace}(\xi_{(P, \bar{c})}^{i+1}) \text{ oder}$$

$$\text{trace}(\xi_{(P, \bar{c})}^i) \xrightarrow{N} \text{trace}(\xi_{(P, \bar{c})}^{i+1})$$

Teil 3: $(P, \bar{c}) \neq \omega \implies$

$$\bigvee_{i \in \mathbb{N}} \xi_{(P, \bar{c})}^i \neq \Omega \text{ und } \text{trace}(\xi_{(P, \bar{c})}^i) \text{ nicht in}$$

$$\text{Normalform} \implies \bigvee_{m \in \mathbb{N}} \text{trace}(\xi_{(P, \bar{c})}^i) \xrightarrow{N} \text{trace}(\xi_{(P, \bar{c})}^{i+m})$$

Eine detaillierte Ausführung des Beweises liegt in /FE/ vor.

3. Schlußbemerkung

P.Landin hat in /LP/ die enge Beziehung zwischen dem λ -Kalkül und prozedurorientierten Programmiersprachen aufgezeigt. Nach dem Korrektheitsbeweis des LPI lassen sich u.a. folgende Ergebnisse auf Interpreter und Compiler höherer Programmiersprachen übertragen:

1. Auf Variablenumbenennung kann verzichtet werden.
2. Wörtliche Substitution wird vermieden durch Interpretation der gebundenen Variablen als Prozeduraufrufe unter Verwendung eines Kellerspeichers (activation-record-stack).
3. Direkter Zugriff zu gültigen Eintragungen in der Symboltabelle über einen Environmentvektor, dessen Komponenten gemäß der "static-chain" geordnet sind, ist korrekt.

Literaturverzeichnis:

- /CA/ Church,A.; The Calculi of Lambda-Conversion, Princeton 1941
- /FE/ Fehr,E.; Eine universelle Lambda-Kalkül-Programmiersprache und
ihr Interpreter, Berichte der Informatik, Band 1, Universität
Bonn,(erscheint)
- /IK/ Indermark, K.; Semantik von Programmiersprachen, Vorlesungs-
manuskript, Bonn 1973
- /LP/ Landin, P.; A Correspondence between ALGOL 60 and Church's Lambda-
Notation, Comm. ACM 8, 283, 1965
- /ML/ McGowan,C.L.; Correctness Results for Lambda-Calculus Interpreters,
Cornell-University, Ph.D., 1971
- /WP/ Wegner, P.; Programming Languages, Information Structures, and
Machine Organisation, McGraw Hill, 1968
- /WV/ Wegner, P.; The Vienna Definition Language, Providence, Rhode
Island, 02912