
Vorbesprechung
Berechnungsmodelle, Lambda-Kalkül
Evolution, Prinzipien hinter feinen Programmstrukturen
Programmier-
paradigmen

Team

Vortragende

Andreas Krall
Franz Puntigam

Tutor_innen

(A) Lukas Briem
(B) Lukas Dichtl
(C) Radina Grancharova
(D) Geoffrey Karnbach
(E) Peter Kotouc
(F) Lukas Lakits
(G) Balint Timar
(H) Elisa Wang

Thema

Paradigma = bestimmte Denkweise oder Weltanschauung

ein Programmierstil ist individuell,
Gesamtheit der Stile, die auf gemeinsamen Denkweisen beruhen, ergibt ein Paradigma

Beispiele:

prozedurale, objektorientierte, funktionale, parallele, nebenläufige Programmierung

Lernergebnisse (kurzgefasst):

Paradigmen voneinander unterscheiden und in Fachterminologie beschreiben,
für Paradigmen typische Vogehensweisen, Techniken, Konzepte praktisch anwenden,
Aufgaben in Programme vorgegebener Stile umsetzen, diese kritisch beurteilen

Methoden

Skriptum (über TUWEL) Skriptum *vor* Lösen der Aufgaben lesen

Vorlesungen (Mo, 15:00 Uhr) und Fragestunden (Do, 15:00 Uhr, ZOOM)

Übung in 3er-Gruppen, fast wöchentlich eine Aufgabe Ø 10–12 Stunden pro Aufgabe/Person

Abgabegespräch zu Übungsaufgaben im Jänner 2024

3 schriftliche Tests (27. 11. 2023, 22. 1. 2024, 8. 3. 2024), 2 davon gewertet

Anmeldung

Anmeldung zur Lehrveranstaltung in TISS (überprüft STEOP)

wenn noch keine vollständige Übungsgruppe mit 3 Personen:

Teilnahme an Gruppenfindungstreffen (online, siehe TUWEL)
oder Suche über Forum (TUWEL) zuerst absprechen, nicht in Zufallsgruppe anmelden

nachdem zu vollständiger Übungsgruppe abgesprochen:

Anmeldung zu Übungsgruppe in TISS Gruppenmitglieder in kurzem Abstand

Anmeldung bis 6. 10. 2023, 14:00 Uhr danach werden Gruppen zugewiesen
Abmeldung über TISS bis 30. 10. 2023, 10:00 Uhr möglich

Benutzung von Rechnern

Server für Übung: `g0.complang.tuwien.ac.at`

benötigt: Account und git-Repository — Zugangsinformationen am 9. 10. per eMail

Arbeiten auf eigenen Rechnern möglich

nötige Software: Java und Git

empfohlene Software: IntelliJ IDEA

Übungsaufgaben

fast jeden Montag ab 9. 10. 2023 neue Aufgabe auf TUWEL (insgesamt 9)

Abgabe = Absammeln der Lösungen aus git-Repository um 14:00 Uhr strikte Deadline

Abgaben nach einer Woche (1. Aufgabe) oder zwei Wochen (meist), meist wöchentlich!
Fristverlängerung bedeutet Punkteabzug regulären Abgabetermin unbedingt einhalten

Beurteilung der Lösungen

Einstiegsphase (3 Aufgaben) von Tutor_in beurteilt nach
Mitarbeit bzw. Vollständigkeit und Funktionalität, zusammen bis 100 Punkte

Aufforderungen zu Nachbesserungen durch Tutor_in fristgerecht nachkommen

weitere 6 Lösungen von Vortragenden beurteilt nach
aufgabenspezifischen Qualitäten, bis 100 Punkte pro Aufgabe Beurteilungskriterien beachten

vorläufige Ergebnisse per eMail

Abgabegespräche im Jänner mit Begründung der Lösungen,
unterschiedliche Beurteilungen für Mitglieder der gleichen Gruppe möglich

mindestens 50% der max. 700 Punkte für positiven Übungsteil nötig

Gesamtbeurteilung

je zur Hälfte Tests zur Theorie und Übungsbeurteilung

positive Note nur wenn Theorieteil *und* Übungsteil positiv

Prüfungsanmeldung in TISS für Testteilnahme nötig für jeden Test einzeln anmelden

100 Punkte pro Test erreichbar, maximal 2 (von 3) Tests zählen (die besseren)

insgesamt mindestens 100 Punkte aus 2 Tests für positive Beurteilung des Theorieteils

Bei Fragen und Problemen

Lehrveranstaltungsbeschreibung in TISS,
Nachrichten- und Diskussionsforum in TUWEL

themenspezifische Fragestunden (ZOOM, Zugang über TUWEL)

eMail an betreuende_n Tutor_in, ständig eMail-Kontakt halten
Besprechungstermin (auch online) mit Tutor_in vereinbaren

eMail an Puntigam oder Krall:

franz.puntigam@tuwien.ac.at
andreas.krall@tuwien.ac.at

Berechnungsmodelle als formale Grundlagen

Berechnungsmodell = formales Modell, das alles Berechenbare ausdrücken kann
Mächtigkeit der Turing-Maschine, vollständige Sprache

Beispiele:

- Funktionen (primitiv rekursiv, μ-rekursiv, λ-Kalkül)
- Prädikatenlogik, Horn-Klauseln
- temporale Logiken, Petri-Netze
- freie Algebren
- Prozesskalküle
- Automaten
- LOOP, WHILE, GOTO, PRAM, ...

λ-Kalkül – Ausdrücke, freie Variablen, Ersetzung

gegeben: unendliche Menge V von *Variablen* (einschließlich Konstanten, Literale)

λ-*Ausdrücke* in E:	$v \in E$ wenn $v \in V$		jedes Element von V ist λ-Ausdruck
Syntax	$f\ e \in E$ wenn $e, f \in E$		Anwendung von f auf e
	$\lambda v.e \in E$ wenn $v \in V; e \in E$		λ-Abstraktion; Parameter v, Ergebnis e

freie Variablen $\mathrm{fv}(e)$ von e:

$$\mathrm{fv}(v) = \{v\} \qquad (v \in V)$$
$$\mathrm{fv}(f\ e) = \mathrm{fv}(f) \cup \mathrm{fv}(e) \qquad \mathrm{fv}(f) \text{ vereinigt mit } \mathrm{fv}(e)$$
$$\mathrm{fv}(\lambda v.e) = \mathrm{fv}(e) \setminus \{v\} \qquad \mathrm{fv}(e) \text{ ohne } v$$

$[e/v]f$ steht für $f \in E$ nach Ersetzung aller freien Vorkommen von $v \in V$ durch $e \in E$

Ersetzung:

$$[e/v]v = e$$
$$[e/v]u = u \qquad (u \in V; u \neq v)$$
$$[e/v](f\ g) = ([e/v]f)\,([e/v]g)$$
$$[e/v](\lambda v.f) = \lambda v.f \qquad v \text{ ist gebundene Variable, nicht frei}$$
$$[e/v](\lambda u.f) = \lambda u.[e/v]f \qquad (u \neq v; u \notin \mathrm{fv}(e)) \text{ vermeidet Namenskonflikt}$$

Regeln des λ-Kalküls

Äquivalenzregeln: ungerichtet

$$\lambda u.e \equiv \lambda v.[v/u]e \quad (v \notin \mathrm{fv}(e)) \qquad \alpha\text{-Konversion (Umbenennung)}$$
$$(\lambda v.f)\, e \equiv [e/v]f \qquad \beta\text{-Konversion (Anwendung)}$$
$$\lambda v.(e\, v) \equiv e \quad (v \notin \mathrm{fv}(e)) \qquad \eta\text{-Konversion (Sonderfall)}$$

$e \equiv f$ wenn $e \in E$ durch (wiederholte) Anwendung obiger Regeln in $f \in E$ umwandelbar

Reduktionsregeln: strikt von links nach rechts

$$(\lambda v.f)\, e \to [e/v]f \qquad \beta\text{-Reduktion}$$
$$\lambda v.(e\, v) \to e \quad (v \notin \mathrm{fv}(e)) \qquad \eta\text{-Reduktion}$$

$e \in E$ ist in **Normalform** wenn e durch Reduktionsregeln nicht weiter reduzierbar

Berechnung im λ-Kalkül: reduziere Ausdruck e zu Normalform f (es gilt $e \equiv f$)

Eigenschaften des λ-Kalküls

nicht jeder λ-Ausdruck $e \in E$ ist zu einer Normalform reduzierbar

Endlosreduktionen unvermeidlich

wenn es zu $e \in E$ einen λ-Ausdruck $f \in E$ in Normalform mit $e \equiv f$ gibt,
dann gibt es auch Reduktionsreihenfolgen, die e zu Normalformen reduzieren,
aber es kann dennoch endlose Reduktionsreihenfolgen geben

Endlosreduktionen von Auswahl der Reduktionsschritte abhängig

sichere Vorgehensweise: jeweils äußersten möglichen Ausdruck für Reduktion wählen

jede Reihenfolge von Reduktionen führt zur gleichen Normalform bis auf α-Konversion
(wenn sie zu einer Normalform führt)

Ergebnis nicht von Reduktionsreihenfolge abhängig

λ-Kalkül ist extrem einfach, aber so mächtig wie die Turing-Maschine

Einfachheit des Kalküls bedeutet nicht, dass Programme einfach sind

Beispiele für Reduktionen im λ-Kalkül

ohne Normalform: $(\lambda v.(v\ v))(\lambda v.(v\ v)) \to (\lambda v.(v\ v))(\lambda v.(v\ v)) \to \cdots$

mehrere Parameter durch geschachtelte λ-Abstraktionen (Currying):
$((\lambda u.(\lambda v.((u\ v)(v\ u))))a)b \to (\lambda v.((a\ v)(v\ a)))b \to (a\ b)(b\ a)$

bedingte Ausführung:

$\lambda u.(\lambda v.u)$	`true`: nimm erstes Argument
$\lambda u.(\lambda v.v)$	`false`: nimm zweites Argument
$\lambda b.(\lambda u.(\lambda v.((b\ u)v)))$	`if_then_else`

$$
\begin{aligned}
&(((\lambda b.(\lambda u.(\lambda v.((b\ u)v))))(\underline{\lambda u.(\lambda v.u)}))a)b \to \\
&((\lambda u.(\lambda v.(((\underline{\lambda u.(\lambda v.u)})u)v)))a)b \to \\
&(\lambda v.(((\underline{\lambda u.(\lambda v.u)})a)v))b \to \\
&((\underline{\lambda u.(\lambda v.u)})a)b \to \\
&(\lambda v.a)b \to \\
&a
\end{aligned}
$$

Forderungen an Berechnungsmodelle

Kombinierbarkeit

Konsistenz

Abstraktion

Systemnähe

Unterstützung

Beharrungsvermögen

Evolution auf Basis von Widersprüchen

Programme brauchen { Flexibilität und Ausdruckskraft / Lesbarkeit und Sicherheit / einfache Verständlichkeit } **Widerspruch!**

Widersprüche ergeben Spannungsfeld, in dem Neues ausprobiert wird

→ neue Erfahrungen verändern persönliche Programmierstile

→ Notwendigkeit zur Zusammenarbeit führt zu gleichartigen Änderungen von Stilen

→ gleichartige Änderungen persönlicher Stile ändern Paradigmen nachhaltig

→ Paradigmen ändern sich langsam und gleichen kurzfristige Strömungen aus

persönliche Stile entwickeln sich mit der Erfahrung auf typische Weise weiter

→ typische individuelle Entwicklungswege sind in Paradigmen schon antizipiert

Prozeduren und strukturierte Programmierung

Prozedur $\simeq$ Funktion mit Schwerpunkt auf Seiteneffekten, die Programmzustand ändern

Strukturierte Programmierung = Programmierung mit Prozeduren aufgebaut aus Kontrollstrukturen nur für *Sequenz, Auswahl* und *Wiederholung*

- $\rightarrow$ mehr ist nicht nötig, um jedes Programmverhalten auszudrücken
- $\rightarrow$ Sequenz, Auswahl und Wiederholung sind als **Denkmuster** hinreichend
  Einfachheit durch strukturierte Programmierung $\neq$ Einfachheit im λ-Kalkül
- $\rightarrow$ Goto und Ähnliches (`break`, `continue`, Ausnahmen) ausdrucksstark, aber unnötig
  fehleranfällig weil strukturierte Denkmuster durchbrochen werden
- $\rightarrow$ Idee hinter strukturierter Programmierung in jedem Paradigma übernommen

Funktionen als Daten

primitiv rekursive Funktionen (operieren nur auf elementare Daten) sind nicht vollständig,
μ-rekursive Funktionen werden durch zusätzliche „Kontrollstruktur“ vollständig,
einfacher λ-Kalkül ist durch Verwendung von λ-Abstraktionen als Daten vollständig

→ Funktionen als Daten machen Programme ausdrucksstark

→ Funktionen als Daten verschleiern Kontrollfluss (da Daten Kontrollfluss enthalten),
Denkmuster können wesentlich komplizierter sein als bei strukturierter Programmierung

→ Funktionen als Daten ermöglichen **funktionale Formen** statt Kontrollstrukturen,
wie Kontrollstrukturen passen funktionale Formen zur strukturierten Programmierung
Funktionen nur zusammen mit gut durchdachten funktionalen Formen als Daten verwenden

⇒ Abwägung: Freiheit (= Ausdrucksstärke) gegen leicht überschaubare Denkmuster

Objekte als Daten

sehr ähnliche Argumentationskette wie oben (Objekte statt Funktionen als Daten),
aber **Abstraktionen des Objektverhaltens** statt funktionaler Formen

→ komplexe Denkmuster, die Abstraktionen des Objektverhaltens einbeziehen

→ objektorientierte Denkmuster gänzlich verschieden von funktionalen,
auch wenn Techniken (Funktionen/Objekte als Daten) einander ähneln können

→ wenn Objekte als Daten verwendbar, dann indirekt auch Funktionen
Mechanismus für entweder Funktionen oder Objekte als Daten, nicht beides

→ je bessere Unterstützung für Funktionen/Objekte als Daten, desto weniger effizient

⇒ Abwägung: Abstraktion gegen Ausdrucksstärke gegen wenige einfache Denkmuster
objektorientierte gegen funktionale gegen prozedurale Programmierung