Lambda

Äquivalenzregeln:	$\lambda u.e \equiv \lambda v.[v/u]e$	$(v \notin \mathrm{fv}(e))$	α-Konversion (Umbenennung)
ungerichtet	$(\lambda v.f)\, e \equiv [e/v]f$		β-Konversion (Anwendung)
	$\lambda v.(e\, v) \equiv e$	$(v \notin \mathrm{fv}(e))$	η-Konversion (Sonderfall)

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

Reduktionsregeln:	$(\lambda v.f)\, e \to [e/v]f$		β-Reduktion
strikt von links nach rechts	$\lambda v.(e\, v) \to e$	$(v \notin \mathrm{fv}(e))$	η-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 $e \equiv f$)

- nicht jeder λ-Ausdruck e ∈ E ist zu einer Normalform reduzierbar (Endlosreduktionen unvermeidlich)
- jede Reihenfolge von Reduktionen führt zur gleichen Normalform bis auf α-Konversion (wenn sie zu einer Normalform führt)
- λ-Kalkül ist extrem einfach, aber so mächtig wie die Turing-Maschine

Modularisierung

Objekt

- existiert zur Laufzeit
- anonym
- Identität
- Zustand und Verhalt unterscheidbar

Klasse

- für Objekterzeugung
- Klassifizierung (Java-Klasse/Interface)
- nicht static

Module

- Übersetzungseinheit
- importiert über Namen
- zyklenfrei Java-Klasse
- static

Komponente

- Übersetzungseinheit
- Import anonym
- Deployment nötig; z. B. Java-EE-Bean

Namensraum

- zur Organisation von Modularisierungseinheiten; z. B. Java-Paket

Parametrisierung

Parametrisierung = späteres Befüllen von in Modularisierungseinheiten belassenen Lücken

für Objekte durch Setzen von Objektvariablen, wobei Werte so in Objekte gelangen:

- Konstruktor: übliche Vorgehensweise
- Initialisierungsmethode: z. B. bei zyklischen Abhängigkeiten oder zusammen mit clone()
- zentrale Ablage: Objekt holt sich Werte selbst von vorbestimmten Plätzen

für Klassen, Module, Komponenten: oft keine „normalen“ Werte zum Befüllen der Lücken

- Generizität: Lücken z. B. durch Typparameter gekennzeichnet, später durch Typen ersetzt
- Annotationen: Werte an Programmteile angeheftet, von Werkzeugen explizit ausgelesen
- Aspekte: spezifizieren Modifikationen von Programmteilen (zusätzlich ausgeführter Code)

Problem: starke Abhängigkeit zwischen Lücken und Werten zum Befüllen

Paradigmen

Prozedurales Paradigma

- Programmfluss soll Kontrollfluss entsprechen
- globale Variablen und Aliase erlaubt und nötig, aber unerwünscht
- zur Modularisierung höchstens Module
- Prozeduren/Objekte nicht als Daten
- globale Daten, jeder Wert nur auf eine Weise zugreifbar
- schon kleine Programme wirken komplex
- überschaubare, anfängerfreundliche Menge sprachlicher Ausdrucksmittel
- viel Kontrolle über Details
- spezielle Hardware ansprechbar
- schon kleine Programme wirken komplex
- niedrige Abstraktionsgrade, häufig λ-Abstraktion, manchmal nominale Abstraktion
- Basis für formale Korrektheitsbeweise bei λ-Abstraktion
- Schleifen häufig, Rekursion selten
- entweder nur dynamisch oder weitgehend statisch typisiert (explizite Typspezifikationen)
- unkontrollierbare Kommunikation über Variablen und Aliase ist problematisch
- Einsatzgebiete
    - hardwarenahe Programmierung
    - Echtzeitprogrammierung
    - Scripting
    - flexible Software-Architekturen (z. B. Micro-Services)
    - Hobby-Programmierung und Anwendungsprogrammierung
- Hauptziel: gute Kontrolle
- Wichtigste Daten: Zahlen, Arrays
- Programmieren im Feinen
- Abstraktionsform: λ. nominal
- Abstraktionslevel: niedrig
- Seiteneffekte: wichtig
- Umgang mit Aliasen: problematisch
- Erlernbarkeit: einfach
- Fehleranfälligkeit: hoch
- Typisierung: statisch, dynamisch
- Besonders geeignet für: hardwarenahe Programmierung
- nicht geeignet für: große Projekte

Objekt orientiertes Paradigma

- Prozeduren zusammen mit Objekten als Daten
- Objekte, Variablen haben nominale abstrakte Datentypen, abstrakt verständlich
- dynamisches Binden erzwingt abstraktes Verständnis (Kontrollfluss zu unklar)
- örtlich eingegrenzte Kommunikation über Variablen (private)
- offensiver Umgang mit Aliasen (Identität versus Gleichheit)
- verschiedene Daten zu Objekt zusammengefasst
- professionelle Werkzeugkette für Entwicklung und Wartung großer, langlebiger Software
- nominale Abstraktionen auf hohem Niveau

- Zusammenarbeit professioneller Entwickler_innen notwendig
- komplexes Gefüge an Denkmustern
- sehr teuer, aber auch für sehr komplexe Systeme erfolgversprechend
- erfordert vollen Einsatz und viel Wissen („ein bisschen objektorientiert“ ist sinnlos)
- ungeeignet für kleine Projekte und sehr komplexe Algorithmen
- überfordert unerfahrene Programmierer_innen
- Hauptziel: langfristige Wartung
- Wichtigste Daten: Objekte
- Programmieren Groben
- Abstraktionsform: nominal
- Abstraktionslevel: hoch
- Seiteneffekte: wichtig
- Umgang mit Aliasen: Identität
- Erlernbarkeit: schwer
- Fehleranfälligkeit: mittel
- Typisierung: stark
- Besonders geeignet für: große Projekte
- nicht geeignet für: kleine Programme, komplexe Algorithmen

Funktionales Paradigma

- Funktionen als Daten
- ersetzen Kontrollstrukturen
- Modularisierung wichtig
- große Strukturen
- änderbare Daten referenzieren stabile
- Programmierung im Groben gut unterstützt
- ohne Seiteneffekte keine Kommunikation über gemeinsame Variablen
- Aliase harmlos, Original und Kopie nicht unterscheidbar (referenzielle Transparenz)
- „sauber“: aufgesammeltes Wissen geht nie verloren
- Funktion höherer Ordnung = funktionale Form, kann jede Kontrollstruktur ersetzen
- bei hohen Abstraktionsgraden eher λ-Abstraktion oder strukturelle Abstraktion
- ausschließlich Rekursion statt Schleifen
- heute meist vollständig statisch typisiert (Typinferenz)
- Lazy-Evaluation einfach
- Hauptziel: Programmiereffizienz
- Wichtigste Daten: Funktionen
- Programmieren im Feinen
- Abstraktionsform: strukturell, λ, nominal
- Abstraktionslevel: niedrig bis hoch
- Seiteneffekte: verboten
- Umgang mit Aliasen: referentielle Transparenz
- Erlernbarkeit: mittelmäßig
- Fehleranfälligkeit: niedrig
- Typisierung: statisch (Typinferenz)
- Besonders geeignet für: komplexe Algorithmen
- nicht geeignet für: hardwarenahe Programme

Paralleles Paradigma

- kurze Gesamtlaufzeiten auf vielen Prozessoren angestrebt
- Daten meist in Bereiche aufgeteilt, die unabhängig bearbeitbar sind
- Zielerreichung durch Speedup ausgedrückt: Sp = T1/Tp (größer ist besser)
- Speedup abhängig von Details der Aufgabenstellung, Daten und Hardware
- Wissen über diese Details nötig
- Summe der Rechenzeit (p Recheneinheiten) höher als sequentielle Zeit T1 (daher Sp < p)
- Sp > 1 nur wenn Parallelausführung Zusatzaufwand überkompensiert
- hoher Aufwand auf vielen Ebenen (Hardware, Softwareentwicklung, Wartung, . . .)
- zahlt sich nur zur Verarbeitung großer Mengen einheitlich strukturierter Daten aus
- wichtig ist Finden einer Vorgehensweise, die unabhängige Datenblöcke ermöglicht
- fertige Bibliotheken für häufige Einsatzzwecke

Nebenläufiges Paradigma

- effiziente Reaktionsfähigkeit auf Ereignisse angestrebt
- unterschiedliche Handlungsstränge sollen auf unterschiedliche Daten zugreifen
- viele gleichzeitig/überlappt ablaufende Handlungsstränge
- um Programm zu vereinfachen
- Ausführung der Handlungsstränge häufig durch Warten auf Ereignisse unterbrochen
- vielfältige Ziele und Anwendungsgebiete, z. B. Webserver, Telefonanlage, Simulation
- Bewältigung vieler Handlungsstränge (hohe Last) meist wichtiger als kurze Antwortzeiten
- Anzahl der Handlungsstränge an Bedarf, nicht an Hardwarefähigkeiten ausgerichtet
- in Java Handlungsstränge meist als Threads implementiert
- Zugriffe auf gemeinsame Daten kaum vermeidbar, Synchronisation wichtig
- Vermeidung von Liveness-Problemen notwendig, aber schwierig
- zahlreiche Synchronisationsmechanismen, in Java hauptsächlich Monitore

Applikatives Paradigma

- Variante der funktionalen Programmierung
- funktionale Formen mit Hilfsfunktionen parametrisiert und zusammengefügt
- verwendet vorgefertigte Teile mit wenig zusätzlichem Code
- sehr produktiv
- für komplexe Algorithmen geeignet
- kurze, kompakte Programme mit wenigen Funktionen
- Rekursion kaum sichtbar, nur im Hintergrund
- beruht auf überschaubarer Menge zusammenpassender vorgefertigter funktionaler Formen
- kann div. Programmiertechniken einfach unterstützen (Lazy-Evaluation, Parallelität, . . .)
- Programmstruktur kann sehr kreativ sein
- hoher Abstraktionsgrad, strukturelle Abstraktion oder λ-Abstraktion
- meist generisch und statisch typisiert mit Typinferenz
- eher schwer lesbar

Liveness - Probleme

Starvation

- wichtige Programmteile bekommen nicht genug Ressourcen (etwa Rechenzeit)
- Auswirkung: langsamer Programmfortschritt, kann zum Stillstand führen
- Ursache: unwichtige Programmteile binden Ressourcen (schlechte Ressourcenverteilung)
- Erkennen des Problems: ausgiebig Testen
- Problembeseitigung: gezielte Steuerung, z. B. Zwischenschalten passender Puffer

Deadlock

- mehrere Threads warten gegenseitig auf exklusive Objektzugriffe, die sie halten
- Auswirkung: gegenseitige dauerhafte Blockade, kein Programmfortschritt
- Ursache: mehrere Threads halten und brauchen exklusive Zugriffe auf gleiche Objekte
- Erkennen des Problems: ausgiebig Testen, Programmanalyse (statisch oder dynamisch)
- Problembeseitigung: Timeout, exklusiver Zugriff nur in vorbestimmter Reihenfolge

Livelock

- ähnelt Deadlock, statt Warten wird nachgefragt, ob exklusiver Zugriff möglich
- Auswirkung und Ursache wie bei Deadlock
- Erkennen des Problems: ausgiebig Testen, Programmanalyse kaum zielführend
- Problembeseitigung: Timeout, nicht aktiv warten, nicht „schrittweise Anfragen“

Ersetzbarkeit und Untertypen

Ersetzbarkeit

- U ist Untertyp von T wenn eine Instanz von U überall verwendbar ist wo Instanz von T erwartet wird
- Ersetzbarkeit muss Verhalten berücksichtigen → Design by Contract

Arten von Abstraktionshierarchien

- Untertypbeziehung
  - B ist Untertyp von A wenn jede Instanz von B verwendet werden
  - Ersetzbarkeit wichtig
- Vererbungsbeziehung
  - Übernehmen von Programmtexten aus Oberklasse in Unterklasse
  - Ersetzbarkeit egal
- Untertypbeziehung höherer Ordnung
  - wenn B⟨U⟩ Untertyp von A⟨U⟩ für alle U

Varianz von Typen

- Kovarianz
  - A verhält sich zu B wie T zu U
  - Typ von Konstante, Ergebnis, Ausgangsparameter
- Kontravarianz
  - A verhält sich zu B umgekehrt wie T zu U
  - Typ von Eingangsparameter
- Invarianz
  - gleichzeitig Ko- und Kontravarianz
  - Typ von Variable, Durchgangsparameter

Zusicherungen

- Client-Server-Beziehungen
  - Server bietet Dienste an, Client nutzt Dienste
- Vorbedingung (Precondition)
  - Wer: Client
  - Wann: vor Methodenaufruf.
  - Was: hauptsächlich Eigenschaften von Argumenten
- Nachbedingung (Postcondition)
  - Wer: Server
  - Wann: vor Rückkehr von Methodenaufruf
  - Was: Eigenschaften von Methodenergebnissen sowie Änderungen und Eigenschaften des Objektzustands
- Invariante
  - Wer: Server
  - Wann: vor und nach Ausführung von Methoden
  - Was: unveränderliche Eigenschaften von Objekten und Variablen
- Server-kontrollierter History-Constraint
  - Wer: Server

    - Wann: nach Ausführung von Methoden
    - Was: Einschränkungen auf Veränderungen von Variable
- Client-kontrollierter History-Constraint
    - Wer: Client
    - Wann: vor Methodenaufrufen
    - Was: Einschränkungen auf der Reihenfolge von Aufrufen

Zusicherungen mit Untertypen

- Vorbedingungen in Untertypen sind schwächer oder gleich
- Nachbedingungen in Untertypen sind stärker oder gleich
- Invarianten in Untertypen sind stärker oder gleich

Faustregeln zu Zusicherungen

Zusicherungen sollen

- stabil sein (vor allem an Wurzel der Typhierarchie)
- keine unnötigen Details festlegen
- explizit im Programm stehen
- unmissverständlich formuliert sein
- während Programmentwicklung ständig überprüft werden

Sichtbarkeitsregeln

	`public`	`protected`	—	`private`
sichtbar im selben Paket	ja	ja	ja	nein
sichtbar in anderem Paket	ja	nein	nein	nein
ererbbar im selben Paket	ja	ja	ja	nein
ererbbar in anderem Paket	ja	ja	nein	nein

Ausnahmebehandlungen

Ursachen unvorhergesehener Programmabbrüche finden (kaum vermeidbar): gut

kontrolliertes Wiederaufsetzen nach Fehlern (notwendig, aber schwierig): gut

vorzeitiger Ausstieg aus Sprachkonstrukten (fehleranfällig, vermeidbar): schlecht

Rückgabe alternativer Ergebniswerte (schlechte Programmstruktur, vermeidbar): schlecht