Einführung in die Programmiertechnik

Funktionale Programmierung: Scheme

Grundlagen funktionaler Programmierung

- Idee: Zu lösendes Problem wird als mathematische Funktion formuliert
  - Beispiel Rechtschreibprüfung: Gegeben das zu prüfende Programm, bestimme die Menge aller Schreibfehler (leere Menge: Dokument ist fehlerfrei)
  - Beispiel Wettervorhersage: Gegeben das aktuelle Wetter, bestimme das Wetter in 24h
  - Beispiel automatisches Übersetzen: Gegeben einen englischen Text, ermittle einen entsprechenden russischen Text
- Definition der Funktion erfolgt nicht durch Aufzählung von Rechenschritten, sondern durch Zusammensetzung aus vorhandenen Funktionen
  - Beispiel Rechtschreibprüfung: Teilfunktion könnte z.B. “Gegeben ein Wort, bestimme den Wortstamm” sein

Programmelemente

- Verzicht auf explizite Programmsteuerung (Schleifen, bedingte Anweisungen)
- Statt dessen:
  - Funktion
  - Ausdruck
  - Wert
- getypte oder ungetypte Sprachen
  - ungetypte Sprachen: Alle Werte gehören zu einem Typ
  - auch: statisch oder dynamisch getypte Sprachen
- Funktion i.d.R. partiell für Parametertypen
  - formal spezieller Wert $\perp$ für “kein Wert”
- Datentypen

Striktheit

- Auswertungsreihenfolge: Werden für eine Funktion erst die Argumente ausgewertet und dann die Funktionsdefinition angewendet?
  - nicht-strikte Sprachen: Auswertungsreihenfolge ist nicht festgelegt; Funktionsdefinition wird u.U. angewendet, ohne alle Argumente berechnet zu haben
- Beispiel: (6*5 < 20) & (8/4 < 2)
  - Zur Ermittlung des Gesamtergebnisses (#f) reicht die Ermittlung des ersten Arguments von &
- Bewertungsstrategien (*evaluation strategies*):
  - Regeln, wann Argumente ausgewertet werden
  - z.B. parallel Auswertung, *lazy evaluation*, ...

Striktheit (2)

- Striktheit und ⊥:
  - strikte Auswertung: Falls ein Argument ⊥ ist, ist das Ergebnis ⊥
  - nicht-strikte Auswertung: Falls ein Argument ⊥ ist, kann das Ergebnis trotzdem wohl-definiert sein
    - Beispiel: (20 > 5) | (10/0 < 7) kann als T ausgewertet werden, falls Definition lautet:
      X & Y ist #t falls X den Wert #t hat oder Y den Wert T hat
- Striktheit und Terminierung
  - falls die Auswertung eines Arguments nicht terminiert, terminiert bei strikter Auswertung auch das Ergebnis nicht
  - bei nicht-strikter Auswertung hängt es von der Strategie ab
  - Nicht-terminierende Berechnungen werden oft als gleich zu ⊥ betrachtet

LISP

- *LIst Processing Language*
- Erfunden 1958 von McCarthy am MIT
  - CACM, “Recursive Functions of Symbolic Epressions and Their Computation by Machine”
- Erste Implementierung auf IBM 704
  - 36-bit Rechner (erster Rechner mit Gleitkommahardware)
  - Hauptspeicher auf Basis von Ferritkernen (*core memory*)
  - 40 000 Anweisungen pro Sekunde
  - zwischen 1955 und 1960 wurden 123 Geräte verkauft
- viele Sprachversionen
- 1994 Definition von ANSI Common Lisp (CL)

Scheme

- LISP-Dialekt
- entwickelt in den 1970er Jahren
  - Guy Steele und Gerald Sussmann
  - "Lambda Papers"
- minimalistisch
- aktuelle Version: R^{6}RS (2007)
  - Revised^6 Report on the Algorithmic Language Scheme
  - Unicode-Bezeichner; Bezeichner dürfen mit "->" beginnen
  - Modularisierung der Basisbibliothek
  - etliche neue Prozeduren
  - denotationale Semantik ersetzt durch operationale Semantik
  - ...

Scheme (2)

- strikte Sprache
- Syntax basiert auf S-Ausdrücken (*S-expressions*)
- statische Variablensichtbarkeit
- feste Menge vordefinierter Datentypen
  - boolean
  - Zahlen (exakt, unexakt) (ganz, rational, komplex)
  - Zeichen, Strings
  - Symbole
  - Listen, Hashtables
  - ...
- vordefinierter Funktionen
- imperative Konstrukte (keine “reine” funktionale Sprache)
- erweiterbare Syntax

S-Ausdrücke

- “symbolic expressions”, pairs
- textuelle Notation für Datenstrukturen
- Zwei Formen:
  - ATOM
  - (sexp . sexp)
    - Teile heißen car und cdr
- Atome: Zahlen, Symbole, ...
  - Symbole: Namen, ohne Unterscheidung von Groß- und Kleinschreibung (R6RS: case sensitive)
  - spezielles Symbol: null
- Beispiel: (1 . 2)
- Beispiel: (a . (b . (c . null)))
  - Kurzform falls “letzter” cdr null ist: (a b c)

PLT Scheme

- entwickelt von Matthias Felleisen
  - www.plt-scheme.org
- Entwicklungsumgebung: DrScheme
  - read-eval-print loop (*repl*)
  - Quelltexteditor, Debugger, ...
- Interpreter-Modus: mzscheme *quelldatei*
  - -i geht nach Einlesen/Ausführung in den interaktiven Modus
  - -e <ausdruck>: Auswertung und Ausgabe von Ausdrücken
  - Dateiendung für Quelldateien: .ss
- Compiler-Modus: mzc
  - Ergebnis: Byte-Code-Datei
  - --exe: Erzeugung selbständiger Programme

Ausdrücke

- Syntax: (funktion argumente)
  - (+ 3 4)
  - (+ 10 6 3 9 5)
  - (> 20 5)
    - Logische Wahrheitswerte: #t und #f
  - (max 6 3 9 10 -4)
  - (display (/ 100 3))
    - display ist Funktion mit Seiteneffekt
- Bedingter Operator: Funktion if (nicht strikt)
  - (if (> 10 4)
        (+ 3 5)
        (- 9 2))

Definitionen

- (define variable wert)
- keine Variablen im eigentlichen Sinne
  - Wert nicht nachträglich änderbar
  - aber: nicht-funktionale Zuweisung mittels set!

Funktionen

- Lambda-Ausdrücke
  - (lambda (parameter) koerper)
- Funktionen: benannte Lambda-Ausdrücke

```
(define fib (lambda (n)
   (if (< n 2)
      1
      (+ (fib (- n 1)) (fib (- n 2))))))
```

- Kurznotation ohne explizites lambda
  - (define (*name parameter*) *koerper*)

Variablen

- Namen (Symbole) erhalten Werte bei ihrer Definition; Bindung des Namens nachträglich nicht änderbar
  - Funktionsparameter erhalten Wert bei Funktionsruf
  - let-Konstrukt hilft, weitere Symbole als Abkürzungen einzuführen:
  - (let ((a (fib 10)) (b 8))
    (+ a b))

Listen

- Bestehend aus cons-Zellen (car . cdr)
  - cdr ist wieder Liste
  - Ende der Liste: spezieller Wert NIL
- Zahlreiche vordefinierte Funktionen
  - (first L) liefert erstes Element
    - ehemals: (car L)
  - (rest L) liefert Restliste
    - ehemals: (cdr L)
  - (list-ref N L) liefert N-tes Element (Zählung beginnt mit 0)
    - Spezialfälle: second, third, ... ninth
  - (length L) liefert Länge der Liste
  - (cons V L) liefert Liste, die mit V anfängt und mit L
  - (append L1 L2) liefert Liste, die alle Elemente aus L1 und L2 enthält

quote

- (x y z) bedeutet i.d.R.: rufe Funktion x mit Argumenten y und z
  - x muss Funktionsname sein (oder an Funktion gebundenes Symbol)
  - y und z müssen Symbole sein, die an Werte gebunden sind
- quote: Verwendung von Listen und Symbolen “direkt”
  - (quote (1 2 3))
  - (quote (x y z))
- Kurzform: ‘(x y z)
- Auch für Symbole: ‘x ist Symbol X, nicht Wert von X
  - (list ‘A ‘(B C) (+ 2 3))
- quasiquote: erlaubt, Teile der Liste per unquote zu berechnen
  - `(a b ,x)

Prädikate

- Funktionen, die #t oder #f liefern
- Konvention: Funktionsname endet mit ?
- Typtests: pair?, list?, number?, string?, procedure?
- Tests für Zahlen: even?, odd?, zero?, positive?, exact?, inexact?
- Tests für Zeichen: char-lower-case?, char-alphabetic?
- Tests für Listen: null?
- Tests auf Gleichheit: eq?, eqv?, equal?, char=?, string=?

Strukturen

- Typen mit einer festen Menge benannter Felder
- Definition mithilfe von defstruct
  - (define-struct point x y z)
  - Optional: Initialwerte für Felder, Funktion für Textrepräsentation, ...
- Strukturdefinition definiert implizite Funktionen
- Konstruktor: (make-*struktur werte*)
  - Liste aus Werten
  - (make-point 10 7 4)
- Zugriffsfunktionen: (*struktur-feld wert*)
  - (point-z p) ; p muss point sein
- Typtest: *struktur*?

Funktionen höherer Ordnung

- Funktionen als Parameter von Funktionen
- Beispiel: Gegeben sei Funktion
  - (define (verdoppeln n) (* 2 n))
- Anwenden der Funktion auf alle Elemente einer Liste:
  - (map1 verdoppeln ‘(3 5 10))
- Definition der Funktion map1:

```
(define (map1 f L)
    (if  (null? L)
         null
         (cons (apply f (first L)) (map1 f (rest L)))))
```

- Funktion apply: (apply *f argumente*)
  - Aufruf von Funktion f mit angegebenen Argumenten

Vordefinierte Funktionen höherer Ordnung

- (map *funktion argumentlisten*)
  - Anwendung der funktion auf alle Elemente der Argumentlisten
  - (map + ‘(1 2 3) ‘(4 5 6))
- (andmap *predicate argumentlisten*)
  - Wie map, dann Verknüpfung der Argumente mittels and; Abbruch bei erstem #f; analog: ormap
  - (andmap positive? '(1 43 7 80 2))
- (filter *predicate liste*)
  - Liste aller Argumente, für die Prädikat erfüllt ist
  - (filter even? ‘(1 2 4 8 16 17))
- (sort *liste less-than?*)
  - Sortiere Liste nach Kriterium less-than?
  - (sort '("Hallo" "Welt") [lambda (a b) (< (string-length a) (string-length b))])

Imperative Konzepte in Scheme

- Seiteneffekte
  - Globale Variablen
  - Variablenzuweisungen
  - Änderung von Datenstrukturen
  - Ein/Ausgabe
- Kontrollfluss
  - Sequentielle Ausführung (begin, let, ..)
  - bedingte Ausführung (if, when, unless, cond, and, or)
  - Schleifen (for, do, dolist, ...)

Ein-/Ausgabe

- Öffnen von Dateien: (open-input-file “dateiname”)
- Schließen mit close-output-port
- Automatisches Schließen am Ende der Bearbeitung: call-with-input-file
  - (with-open-file *dateiname funktion*)
  - options: :mode *flag,* :exists *action*
  - flag: 'binary, 'text
  - action (nur bei open-output-file): 'error, 'replace, 'truncate, ...
- read: Lesen in Scheme-Syntax
- read-char, read-byte, read-line: low-level

Weitere Konzepte

- *reader* und *printer*
- *continuations*
- *vectors, hash tables, dictionaries*
- Modularisierung: Klassen, *units*, *contracts*
- Syntaxdefinition und Makros
- *errors* und *violations*