Programmiersprachen
Vorlesung 5 vom 15.11.21
Datenabstraktion

Christoph Lüth

Universität Bremen

Wintersemester 2021/22

Organisatorisches

- Erste Vergabe der Referatsthemen am Donnerstag (18.11.2021)
- Bei Interesse gerne vorher Mail an Veranstalter
- Es gilt im Zweifel first come, first serve.
- List der Sprachen ist **nicht exklusiv** — nehme gerne weitere Vorschläge entgegen.

Liste möglicher Sprachen

- Systemnah: Rust
- Logische Programmierung: Prolog, Oz
- Dynamisch: JavaScript
- Nebenläufig/Reaktiv: Erlang, Golang
- Abhängige Typen: Idris, Agfa/Agda, Liquid X (Dependent types)
- Prozedural: Julia, Kotlin, Swift
- Skriptsprachen: Lua, Tcl, sh/bash
- Funktional: SML/OCaml, Elm, Clojure, LISP, Scala
- Stack-basiert: Forth
- Historisch: COBOL, Algol-68, APL, Ada, Smalltalk
- Datenflusssprachen: Id, Lucid, Lustre
- DSLs: R, SQL, Postscript, TeX, Verilog/VHDL, SystemC, SpinalHDL

Wo sind wir?

- Einführung
- Werte und Typen
- Anweisungen, Variablen und Zustand
- Kontrollabstraktion
- Datenabstraktion
- Fortgeschrittene Typsysteme
- Nebenläufigkeit
- Objektorientierung
- Skriptsprachen
- Beispielsprache II
- Ab Woche 11: Studentische Vorträge.

Abstraktion

- Kontrollabstraktion (done):
  - Jenseits der direkten Auswertung
  - Prozeduren und Parameter
  - Sprünge
- Datenabstraktion (heute):
  - Abstrakte Datentypen
  - Verkapselung und Objekte
  - Module
  - Packages

Wozu Datenabstraktion?

- Kontrollabstraktion hilft uns, Programme **verständlich** zu machen.
- Datenabstraktion hilft uns, **große** Programme verständlich zu machen.
- Indem wir existierende Daten und Funktionen (Methoden) zu neuen Datentypen zusammenfassen erlauben wir Abstraktion in der Sprache.
  - Abstraktion = "geordnetes Weglassen", hier: von Implementationsdetails.
  - Triviales Beispiel: `Int` als `Bool`, ist `0` jetzt `True` oder `False`?

Abstrakte Datentypen

Abstrakter Datentyp (ADT)

Ein ADT besteht aus besteht aus einem (oder mehreren) **Typen** und **Operationen** darauf, mit folgenden Eigenschaften:

1. Werte des Typen können nur über die Operationen **erzeugt** werden
2. Eigenschaften von Werten des Typen werden nur über die Operationen **beobachtet**.
3. Die Einhaltung von **Invarianten** über dem Typ kann garantiert werden

- Damit eine Programmiersprache ADTs unterstützt, müssen wir die **Sichtbarkeit** einschränken können (*information hiding*).
- **Repräsentationsunabhängigkeit**: Eigenschaften sollten von konkreter Implementation unabhängig sein.

Beispiel: Stack

Ein Stack (von ganzen Zahlen) hat mehrere Operationen:

- den leeren Stack (**empty**),
- eine Zahl auf den Stack schieben (**push**),
- die oberste Zahl vom Stack nehmen (**top** und **pop**),
- und einen Test, ob der Stack leer ist (**isEmpty**).

mit folgenden Eigenschaften:

1. der leere Stack ist leer, und nur der;
2. das oberste Element das Stacks ist das letzte darauf geschobene;
3. wenn ich von einem Stack, auf den ich ein Element geschoben habe, das oberste Element herunternehme, ändert sich nichts.

Stack: Implementation

- Beispiel: Stack in Python
  - Stack als Liste
  - Stack als Feld
- Unveränderliche Stacks (immutable): `push` und `pop` liefern **neuen** Stack
- Veränderliche Stacks (mutable): `push` und `pop` haben Seiteneffekte
- Problem: Python schränkt Sichtbarkeit bedingt ein
  - Keine Trennung zwischen Interface und Implementation
  - Private Felder **syntaktisch** gekennzeichnet (`__name`), Zugriff trotzdem möglich (als `_Class__name`)
  - Python Design-Prinzip: "Wir sind alle erwachsen."
- Beispiel: Stack in C
  - Interface `stack.h` syntaktisch getrennt

Stacks in anderen Sprachen

- In Haskell:
  - Wir können Sichtbarkeit einschränken, aber syntaktisch nicht trennen
  - Nur funktionale Lösung
- In Java:
  - **Interfaces** als separates Konstrukt

Spezifikation

- Wollen Stacks **mathematisch** beschreiben
  - Möglichst unzweideutig
  - Können daraus Tests generieren
- Hier:

$$\text{top}(\text{push}(s, x)) = x \qquad \text{isEmpty}(\text{empty})$$

$$\text{pop}(\text{push}(s, x)) = s \qquad \neg(\text{isEmpty}(\text{push}(s, x)))$$

- Gleichungen gelten nur für **unveränderliche** (zustandsfreie) Stacks
- Was bedeutet Gleichheit?
  - Gleichheit für `int` — bekannt
  - Gleichheit für `stack` — nicht gleich, nur **beobachtbar** gleich

Sprachmittel zur Unterstützung von Datenabstraktion

- Sichtbarkeitseinschränkungen aka. Module
- Syntax zur Beschreibung von Schnittstellen
- Trennung der Schnittstelle von der Implementation

Module

- Ein **Modul** ist die Zusammenfassung mehrerer Definition zu einer Einheit
  - Oft mit Verkapselung — Modul hat definierte **Schnittstelle**
  - Module dienen oft auch zur **getrennten Übersetzung** (aber nicht notwendigerweise)
- Module werden deklariert, definiert und benutzt (importiert).
  - Trennt die Sprache das?
  - Wie erfolgt die Benutzung?
  - Wie verhalten sich Module zu Quelldateien?
  - Wie wird importiert — qualifiziert oder unqualifiziert, immer alles, mit Umbenennung?
- Qualifizierter Import: Bezeichner `f` aus Modul `M` wird zu `M.f`.

Module in Python

- Module sind Quelldateien
- Keine Interface-Definition
- Kaum Sichtbarkeitseinschränkungen
- Keine Datenabstraktion
- Import: mit Namen, qualifiziert/unqualifiziert, alles oder selektiv, Umbenennung möglich

Module in Haskell

- Jede Quelldatei ist ein Modul
- Interface: Sichtbarkeit von Bezeichnern kann eingeschränkt werden
- Aber:
  - algebraische Datentypen und Konstruktoren bleiben erkennbar
  - Typsynonyme sind nicht abstrakt
  - Klasseninstanzen werden immer exportiert
- Datenabstraktion möglich (manchmal umständlich)
- Interface keine separate Datei
- Import: mit Namen, qualifiziert/unqualifiziert, alles oder selektiv, Umbenennung möglich

Module in Java

- Module sind **Klassen** (nicht an Quelldatei gebunden)
- Klassen verkapseln interne Repräsentation, Einschränkung der Sichtbarkeit (`public`, `private`, `protected`)
  - Aber Konstruktoren bleiben erkennbar
- Datenabstraktion möglich (durch Reflektion zu durchbrechen?)
- Interfaces sind separates Konstrukt
- Import: nur ganze Klassen, keine Umbenennung, impliziter Import möglich

Module in C

- Module sind Quelldateien (“translation units”)
- Sichtbarkeitseinschränkungen für Bezeichner (`static`, `extern`)
  - Local and global linkage
- Interfaces sind **per Konvention** separate Dateien (`.h`)
  - Konvention wird durch den Präprozessor ermöglicht
- Datenabstraktion möglich (durch Zeigeroperationen zu durchbrechen)
- Import: immer alles, nur unqualifziert, keine Umbenennung

Packages

- Packages schränken die Sichtbarkeit von Modulen ein.
- Existiert in Java, Python.
- Haskell kennt nur hierarchische Module.
- Wenn Module Quelldateien entsprechen, sind Packages Verzeichnisse.

Sprachneutral Interfaces: IDL

- IDL ist die `Interface Definition Language` der OMG
- Definition der Schnittstelle von Komponenten in CORBA
  - Nicht mehr ganz aktuell
- Syntax an C angelehnt
- Compiler erzeugt aus IDL “Rumpf” in entsprechender Programmiersprache
- Alle Funktionsaufrufe gehen über einen “Broker”

Fallbeispiel: Standard ML

- Standard ML (SML) ist eine strikte, nicht-reine funktionale Sprache
  - Typsystem wie Haskell
  - Referenzen und Ein/Ausgabe eingebaut
- Caml und OCaml sind französische Varianten, F# ist OCaml für .Net
- Freie Implementationen, e.g. SML/NJ: `http://www.smlnj.org/`

Module in SML

- Interfaces sind separates Konzept (`signature`)
- Module (`structure`)
- Parametrisierte Module (`functor`)
- Beispiele:
  - Ordnung und Sortieren (`sort.sml`)
  - Finite Maps (`maps.sml`)
- Vorteile:
  - Sehr flexibel, deklarative Beschreibung der Systemstruktur
  - Kann unübersichtlich werden (insbesondere sharing constraints)

Zusammenfassung

- Datenabstraktion durch **abstrakte Datentypen**:
  - Typ mit Operationen darüber
  - Zugriff nur über definierte Schnittstelle
  - Repräsentationsunabhängigkeit
- Module: **Verkapselung** durch Einschränkung der Sichtbarkeit
  - Was wird verkapselt: Sichtbarkeit der Bezeichner, Typrepräsentation, Konstruktoren?
  - Sind Interfaces explizit oder implizit?
  - Modul = Quelldatei?
  - Wie wird importiert?
  - Qualifizierte Bezeichner `M.f`
- Packages: Sammlungen von Modulen
- Fallbeispiel: Module in SML