Teil VIII

Objekte

38. Knobelaufgabe #23

Harry Hacker hat programmiert; was sein Programm wohl macht?

```
let mutable cur = root
while cur <> null do
  if cur.Left = null then
    accept cur.Key
    cur ← cur.Right
  else
    let mutable pre = cur.Left;
    while pre.Right <> null && pre.Right <> cur do
      pre ← pre.Right;
    if pre.Right = null then
      pre.Right ← cur
      cur ← cur.Left
    else
      pre.Right ← null
      accept cur.Key
      cur ← cur.Right
```

Lisa Lista vermutet, dass es um Binärbäume geht ...

38. Gliederung

38. Überblick

Dieses Kapitel ist dem Studium von Objekten gewidmet.

Bisher: Programmierung im Kleinen.

Jetzt: Programmierung im Großen.

☞ Es gibt im wesentlichen zwei Mechanismen, um größere Programme oder Softwaresysteme zu strukturieren:

- *Modulsysteme* und
- *Objektsysteme*.

Das Modulsystem von Mini–F# ist einfach gestrickt: Verwaltung von Namensräumen. Objektsysteme können auf Prototypen oder auf Klassen basieren. Beide Ansätze schauen wir uns im folgenden an ...

I made up the term 'object-oriented', and I can tell you I didn't have C++ in mind.

— Alan Kay, OOPSLA '97

Object oriented programming is, in some sense, just a programming trick using indirection. It's a trick good programmers have been using for years.

— Bjarne Stroustrup

There are only two things wrong with C++. The initial concept and the implementation.

— Bertrand Meyer

38. Charakteristika — Fortsetzung

Was macht eine objektorientierte Sprache aus?

- *Dynamische Bindung* (engl. dynamic dispatch): Wird eine Operation auf einem Objekt durchgeführt, bestimmt das Objekt selbst, welcher Code ausgeführt wird. Objekte mit derselben Schnittstelle können unterschiedlich implementiert sein. Die Implementierungen der Operationen heißen *Methoden*; der Anwendung einer Operation entsprechend *Methodenaufruf* (engl. method invocation). Machmal sagt man auch, dem Objekt wird ein Nachricht geschickt.
- *Kapselung*: Die interne Repräsentation eines Objekts ist außen nicht sichtbar und kann somit lokal geändert werden.

38. Charakteristika — Fortsetzung

- *Untertypen* (interface inheritance): Auf Schnittstellen läßt sich eine natürliche Untertypbeziehung definieren: ein Objekt läßt sich in einem Kontext verwenden, in dem nur eine Teilmenge der Methoden benötigt wird. Auf diese Weise lassen sich polymorphe Funktionen definieren, die unterschiedliche Objekte uniform behandeln.
- *Vererbung* (implementation inheritance): Vererbung erlaubt es, die Implementierung verschiedener Objekten zu faktorisieren, so dass gemeinsames Verhalten nur einmal implementiert werden muss. Prototypenbasierte Sprachen verwenden *Delegation*, klassenbasierte Sprachen bieten Klassen und *Unterklassen* an.

39. Exkurs

☞ Der Aufruf *monus* (e_1, e_2) wirft eine Ausnahme, wenn e_2 größer ist als e_1.

exception *Insufficient* ***of*** *Nat*

let *monus* $(n_1 : Nat, n_2 : Nat) : Nat =$
  if $n_1 \geqslant n_2$ ***then***
    $n_1 \dot{-} n_2$
  else
    raise $(Insufficient\ n_1)$

39. Kapselung

Zur Erinnerung: Kapselung eines Zustandes.

module *Account* $=$
  let mutable private *funds* $= 0$

  let *deposit* $(amount : Nat) =$
    $funds \leftarrow funds + amount$

  let *withdraw* $(amount : Nat) =$
    $funds \leftarrow monus\ (funds, amount)$

  let *balance* $() = funds$

☞ Die Repräsentation eines Bankkontos, die Speicherzelle *funds*, ist nach außen nicht sichtbar.

☞ *Beobachtung: deposit*, *withdraw* und *balance* gehören konzeptionell zusammen, sind aber nur lose gekoppelt.

39. Schnittstellen

Wunsch: Mehrere Operationen zu einer Einheit zusammenfassen.

type *IAccount* =
  interface
    abstract member *Deposit* : *Nat* → *Unit*
    abstract member *Withdraw* : *Nat* → *Unit*
    abstract member *Balance* : *Nat*
  end

☞ Die Schnittstelle (engl. interface) legt fest, was mit einem Konto, einem Element des Typs *IAccount*, gemacht werden kann:

- Mit Hilfe von *Deposit* kann ein Betrag in ein Konto eingezahlt werden; *Withdraw* erlaubt es, einen Betrag abzuheben.
- Der Kontostand kann mit Hilfe der Eigenschaft *Balance* eingesehen werden.

☞ Die interne Repräsentation eines Kontos ist im Typ *nicht* festgelegt und kann, wie wir sehen werden, sehr unterschiedlich sein.

39. Jargon: Objekte, Methoden und Eigenschaften

Eine Schnittstelle heißt auch Objekttyp; Elemente eines Objekttyps heißen entsprechend *Objekte*.

Funktionen werden zu *Methoden* (engl. methods), andere Werte zu *Eigenschaften* (engl. properties):

- *Deposit* und *Withdraw* sind Methoden;
- *Balance* ist eine Eigenschaft.

39. Objekte

Ein Bankkonto:

```
let lisas : IAccount =
  let mutable funds = 0
  {
    new IAccount with
      member self.Deposit (amount : Nat) =
        funds ← funds + amount

      member self.Withdraw (amount : Nat) =
        funds ← monus (funds, amount)

      member self.Balance =
        funds
  }
```

☞ {*new IAccount with* ...} ist ein Objektausdruck; er erzeugt ein Element des Typs *IAccount*, ein Objekt. Die Methoden und Eigenschaften der Schnittstelle werden implementiert. Das Objekt hat einen internen Zustand, die Speicherzelle *funds*, um den aktuellen Kontostand zu repräsentieren.

39. Aufruf von Methoden

☞ Zugriff auf die Methoden und Eigenschaften mit der Punktnotation.

```
Mini⟩ lisas.Deposit 4711
()
Mini⟩ lisas.Withdraw 815
()
Mini⟩ lisas.Withdraw 2765
()
Mini⟩ lisas.Balance
1131
```

Jargon: dem Objekt *lisas* wird die Nachricht *Deposit* geschickt. In der Definition von *lisas* steht der frei wählbare Bezeichner *self* jeweils für das Objekt selbst, dem Empfänger von Nachrichten.

39. Objekte

Eine alternative Implementierung eines Bankkontos:

```
let ludwigs : IAccount =
  let size         = 10
  let history      = [| for k in 0 .. size − 1 → 0 |]
  let mutable i    = 0
  let next ()      = (i + 1) % size
  {
    new IAccount with
      member self.Deposit (amount : Nat) =
        history.[next ()] ← history.[i] + amount; i ← next ()
      member self.Withdraw (amount : Nat) =
        history.[next ()] ← monus (history.[i], amount); i ← next ()
      member self.Balance =
        history.[i]
  }
```

39. Demo

```
Mini⟩ lisas.Balance
1131
Mini⟩ ludwigs.Deposit 4711
()
Mini⟩ lisas.Withdraw 1000; ludwigs.Deposit 1000
()
Mini⟩ lisas.Balance
131
Mini⟩ ludwigs.Balance
5711
```

☞ Die beiden Objekte verstehen die gleichen Nachrichten; die interne Umsetzung ist aber ganz unterschiedlich.

☞ Jedes Objekt entscheidet selbst, wie es auf eine Nachricht reagiert, sprich welcher Programmcode ausgeführt wird (dynamic dispatch).

39. Objekte und Schnittstellen

☞ Objekte sind normale Werte; sie können insbesondere Argument oder Ergebnis von Funktionen sein.

let *transfer* $(account_1 : IAccount, amount : Nat, account_2 : IAccount) =$
  $account_1.Withdraw\ amount;$
  $account_2.Deposit\ amount$

Wir können eine Überweisung tätigen, auch wenn die beteiligten Objekte unterschiedlich implementiert sind.

```
Mini⟩ (lisas.Balance, ludwigs.Balance)
(131, 5711)
Mini⟩ transfer (ludwigs, 815, lisas)
()
Mini⟩ (lisas.Balance, ludwigs.Balance)
(946, 4896)
```

(☞ Good SE practice: programming against an interface.)

39. „Objektfabriken“

☞ Von einem Bankkonto zu vielen Bankkonten, sprich einer Bank:

```
module TrustMe =
  let account (seed : Nat) : IAccount =
    let mutable funds = seed
    {
      new IAccount with
        member self.Deposit (amount : Nat) =
          funds ← funds + amount

        member self.Withdraw (amount : Nat) =
          funds ← monus (funds, amount)

        member self.Balance =
          funds
    }
```

☞ die Funktion *TrustMe.account* heißt auch *Konstruktor*, da sie Objekte konstruiert.

39. Module und Objekte

Module und Objekte können sinnvoll kombiniert werden.

```
module TrustMe =
  do putline "TrustMe is founded."
  let BIC = 4711    // Bank Identifier Code
  let mutable private no = 0
  let no-of-accounts () = no
  let account (seed : Nat) : IAccount =
    do no ← no + 1
    let mutable funds = seed
    {
      new IAccount with ...
    }
```

Modul: bankspezifische, kontoübergreifende Operationen (*BIC*, Gesamtzahl der Konten: *no*).
Objekt: kontospezifische Operationen (*Deposit* etc).

39. Delegation

Aufgaben können an andere Objekte delegiert werden (Komposition):

```
module TrustMe =
  ...
  exception Limit
  let limit = 1000
  let student-account (seed : Nat) =
    let basic = account seed
    {new IAccount with
        member self.Deposit amount = basic.Deposit amount
        member self.Withdraw amount =
          if amount > limit then raise Limit
                            else basic.Withdraw amount
        member self.Balance = basic.Balance
    }
```

☞ Die Nachrichten *Deposit*, *Withdraw* und *Balance* werden an das Basisobjekt *basic* delegiert.

39. Demo

Mini⟩ *TrustMe.BIC*
val *it* : *Nat* = 4711
Mini⟩ ***let*** *herberts* = *TrustMe.account* 8150
Mini⟩ ***let*** *harrys* = *TrustMe.student-account* 0
Mini⟩ *TrustMe.no-of-accounts* ()
val *it* : *Nat* = 2
Mini⟩ *transfer* (*herberts*, 2000, *harrys*)
Mini⟩ *transfer* (*harrys*, 2000, *herberts*)
uncaught exception : *Limit*

39. Abstrakte Datentypen versus Objekte

Die Zusammenfassung von Operationen zu einer Schnittstelle kennen wir von den abstrakten Datentypen (Teil V).

- Dort: Realisierung einer Schnittstelle mit Hilfe des Modulsystems.
- Hier: Realisierung einer Schnittstelle mit Hilfe des Objektsystems.

☞ Ein ADT kann auch mit dem Objektsystem realisiert werden.

Unterschiede:

- Verschiedene Implementierungen eines ADTs zeigen exakt das gleiche Verhalten; sie sind austauschbar.
- (Modulsystem: eine Schnittstelle wird zu einem Zeitpunkt durch *eine* Implementierung realisiert.)
- Objekte, die die gleiche Schnittstelle unterstützen, zeigen ein verwandtes, nicht aber notwendigerweise das gleiche Verhalten (Standardkonto versus Studierendenkonto).
- (Objektsystem: eine Schnittstelle kann zu einem Zeitpunkt durch *mehrere* Objekte realisiert werden.)

39. Abstrakte Syntax

☞ Der Übersichtlichkeit halber formalisieren wir nur Schnittstellen mit genau 2 Methoden.

Wir erweitern Typen um Schnittstellentypen (engl.interface types).

$d ::= \cdots$	*Deklarationen:*
$\mid$ ***type*** $T =$	Schnittstellentypdefinition ($\ell_1 \neq \ell_2$)
interface	
abstract member $\ell_1 : t_1$	
abstract member $\ell_2 : t_2$	
end	

39. Abstrakte Syntax

Wir erweitern Ausdrücke um Objektausdrücke (engl. object expressions) und Methodenaufrufe (engl. method invocations).

$e ::= \cdots$	*Ausdrücke:*
$\mid \{$ ***new*** T ***with***	Objektausdruck ($\ell_1 \neq \ell_2$)
member $s_1.\ell_1 = e_1$	
member $s_2.\ell_2 = e_2$	
$\}$	
$\mid e.\ell$	Methodenaufruf

☞ $\{$***new*** T ***with*** $\ldots\}$ ist ein anonymes Objekt; $e.\ell$ ist ein Methodenaufruf: dem Objekt e wird die Nachricht ℓ geschickt.

☞ s_1 bzw. s_2 ist ein beliebiger Bezeichner, der den Empfänger der Nachricht repräsentiert; typische Namen: *self* oder *this*.

☞ Ein Objekt ist eine Sammlung von Methoden; die Methoden legen fest, was mit einem Objekt gemacht werden kann. Ein Objekt ist die Summe seines Verhaltens.

39. Statische Semantik — Objekte

Die folgenden Typregeln setzen voraus, dass die Typdefinition

type $T =$
  interface
    abstract member $\ell_1 : t_1$
    abstract member $\ell_2 : t_2$
  end

bekannt ist.
Typregeln:

$$\frac{\Sigma, \{ s_1 \mapsto T \} \vdash e_1 : t_1 \qquad \Sigma, \{ s_2 \mapsto T \} \vdash e_2 : t_2}{\Sigma \vdash \{ \textbf{\textit{new}}\ T\ \textbf{\textit{with}}\ \textbf{\textit{member}}\ s_1.\ell_1 = e_1\ \textbf{\textit{member}}\ s_2.\ell_2 = e_2 \} : T}$$

$$\frac{\Sigma \vdash e : T}{\Sigma \vdash e.\ell_i : t_i} \quad i \in \{1, 2\}$$

39. Dynamische Semantik — Objekte

Wir erweitern Werte um Methodentabellen und Methodenabschlüsse.

$\mu \in \mathsf{Lab} \to_{\mathrm{fin}} \mathsf{Val}$	*Methodentabellen*
$\nu ::= \cdots$	*Werte:*
$\mid \mu$	Objekt
$\mid \langle\!\langle \delta, s, e \rangle\!\rangle$	Methodenabschluss

☞ Die Methodentabelle (engl. method table or dispatch table) bildet Labels auf Werte ab. Ein Methodenabschluss entspricht einem rekursiven Funktionsabschluss *ohne* Parameter.

39. Dynamische Semantik — Objekte

Auswertungsregeln (dynamische Bindung):

$$\delta \vdash \{\ \textbf{\textit{new}}\ T\ \textbf{\textit{with}}\ \textbf{\textit{member}}\ s_1.\ell_1 = e_1\ \textbf{\textit{member}}\ s_2.\ell_2 = e_2\} \Downarrow \{\ell_1 \mapsto \langle\!\langle \delta,\ s_1,\ e_1 \rangle\!\rangle, \ell_2 \mapsto \langle\!\langle \delta,\ s_2,\ e_2 \rangle\!\rangle\}$$

☞ Ein Objekt wertet im Wesentlichen zu sich selbst aus.

$$\frac{\delta \vdash e \Downarrow \mu \qquad \delta', \{s_i \mapsto \mu\} \vdash e_i \Downarrow \nu_i}{\delta \vdash e.\ell_i \Downarrow \nu_i} \qquad \text{mit } \mu(\ell_i) = \langle\!\langle \delta',\ s_i,\ e_i \rangle\!\rangle$$

☞ Die statische Semantik stellt sicher, dass $\ell \in \mathit{dom}\ \mu$. Das Label ℓ wird zur Laufzeit in der zu dem Objekt e gehörigen Methodentabelle nachgeschlagen (dynamic dispatch). Der „*self*-Bezeichner“ s_i wird an das Objekt selbst (engl. self) gebunden.

39. Knobelaufgabe #15 — da capo

Erinnern wir uns an die Lösung der Knobelaufgabe #15:

$$
\begin{array}{l}
\textbf{\textit{let}}\ \mathit{fac}\ (\mathit{self}, n : \mathit{Nat}) : \mathit{Nat} = \\
\quad \textbf{\textit{if}}\ n = 0\ \textbf{\textit{then}}\ 1 \\
\qquad\qquad \textbf{\textit{else}}\ \ \mathit{self}\ (\mathit{self}, n - 1) * n \\
\textbf{\textit{let}}\ \mathit{factorial}\ (n : \mathit{Nat}) : \mathit{Nat} = \\
\quad \mathit{fac}\ (\mathit{fac}, n)
\end{array}
$$

☞ Rekursion wird durch Selbstapplikation simuliert.

39. Knobelaufgabe #15 — da capo

☞ Genau das Gleiche passiert bei einem Methodenaufruf!

```
type Factorial =
  interface
    abstract member factorial : Nat → Nat
  end
let fac =
  {new Factorial with
    member self.factorial (n : Nat) : Nat =
      if n = 0 then 1
               else self.factorial (n − 1) ∗ n
  }
let factorial (n : Nat) : Nat =
  fac.factorial n
```

39. „Getter“ und „Setter“

```
type IAccount =
  ...
  abstract member Balance : Nat

let lisas : IAccount = ...
  {new IAccount with ...
     member self.Balance = funds
  }
```

Die Definition der Eigenschaft *Balance* ist eine Abkürzung für:

```
type IAccount =
  ...
  abstract member Balance : Nat with get

let lisas : IAccount = ...
  {new IAccount with ...
     member self.Balance with get () = funds
  }
```

39. „Getter“ und „Setter“: Sichten

```
type Person =
  interface
    abstract member Name : String with get
    abstract member Age  : Nat    with get, set
  end

let doctor name alias age =
  let mutable age = age
  {
    new Person with
      member self.Name
        with get () = "Doctor " ^ name ^
                      if alias = "" then ""
                                    else " (aka " ^ alias ^ ")"
      member self.Age
        with get ()  = age
        and  set inc = age ← age + inc
  }
```

39. „Getter“ und „Setter“: Demo

```
Mini⟩ let dolittle = doctor "Dolittle" "King Jong Thinkalot" 42
Mini⟩ dolittle.Name
"Doctor Dolittle (aka King Jong Thinkalot)"
Mini⟩ dolittle.Age ← 2
()
Mini⟩ dolittle.Age
44
```

☞ Die Eigenschaft *Name* kombiniert „Rohdaten“; sie definiert eine Sicht auf die Daten.

☞ Die Zuweisung *dolittle.Age* ← 2 *erhöht* das Alter um den angegebenen Wert (C: `dolittle.Age += 2`.)

39. Vertiefung: Ausdrücke

☞ Objekte müssen nicht notwendigerweise einen internen Zustand haben.

☞ Alternative Darstellung von Ausdrücken:

```
type IExpr =
  interface
    abstract member Value : Nat
    abstract member Show : String
  end
```

☞ Was kann/will ich mit einem Ausdruck machen? Auswerten und in einen String überführen!

☞ Ein Objekt ist die Summe seines Verhaltens.

39. Demo

Mini⟩ ***let*** $e = add\ (constant\ 4711, mul\ (constant\ 815, constant\ 2765))$
Mini⟩ $e.Show$
```
"(4711 + (815 * 2765))"
```
Mini⟩ $e.Value$
2258186
Mini⟩ $(add\ (e, e)).Value$
4516372
Mini⟩ $(add\ (e, e)).Show$
```
"((4711 + (815 * 2765)) + (4711 + (815 * 2765)))"
```

39. Vertiefung: Ausdrücke

```
let constant (n : Nat) : IExpr =
  { new IExpr with
      member self.Value = n
      member self.Show = show n
  }
let add (expr_1 : IExpr, expr_2 : IExpr) : IExpr =
  { new IExpr with
      member self.Value = expr_1.Value + expr_2.Value
      member self.Show = "(" ^ expr_1.Show ^ " + " ^ expr_2.Show ^ ")"
  }
let mul (expr_1 : IExpr, expr_2 : IExpr) : IExpr =
  { new IExpr with
      member self.Value = expr_1.Value * expr_2.Value
      member self.Show = "(" ^ expr_1.Show ^ " * " ^ expr_2.Show ^ ")"
  }
```

☞ *constant*, *add* und *mul* heißen auch *Konstruktoren*, da sie Objekte konstruieren (so wie Datenkonstruktoren Daten konstruieren).

39. Variantentypen versus Objekttypen

Objekte definieren sich durch das Verhalten: was kann mit dem Objekt gemacht werden (Idee des abstrakten Datentyps).

Im Gegensatz zu Varianten- und Recordtypen, die durch die Angabe der Elemente definiert werden (konkreter Datentyp).

Variantentyp: Wie ist ein Ausdruck aufgebaut?

- Einfach: neue Funktionalität hinzufügen. Es wird einfach eine neue Funktion definiert.
- Schwierig: neue Datenkonstruktoren hinzufügen. *Jede* bestehende Funktion muss erweitert werden.

Objekte: Was mache ich mit einem Ausdruck?

- Einfach: neue Objektkonstruktoren hinzufügen. Es wird einfach eine neue Konstruktorfunktion definiert.
- Schwierig: neue Funktionalität hinzufügen. *Jede* bestehende Konstruktorfunktion muss erweitert werden.

39. Generische Schnittstellen

☞ Wie Datentypen können auch Schnittstellen mit einem oder mehreren Typen parametrisiert werden.

```
type IStack ⟨'elem⟩ =
  interface
    abstract member Push : 'elem → Unit
    abstract member Pop  : Unit → 'elem
    abstract member Top  : 'elem
  end
```

☞ Stacks können Elemente beliebigen Typs enthalten; jeder einzelne Stack ist aber homogen: die Elemente besitzen den gleichen Typ.

39. Polymorphe Objektkonstruktoren

```
exception Pop
exception Top

let stack ⟨'elem⟩ () =
  let mutable stack = []
  {
    new IStack ⟨'elem⟩ with
      member self.Push x =
        stack ← x :: stack

      member self.Pop () =
        match stack with
        | []      → raise Pop
        | x :: xs → stack ← xs; x

      member self.Top =
        match stack with
        | []      → raise Top
        | x :: xs → x
  }
```

39. UPN-Taschenrechner Deluxe

```
let upn-calculator-deluxe () =
  let stacks : IStack ⟨IStack ⟨Nat⟩⟩ = stack ()
  try
    stacks.Push (stack ())
    while true do
      try
        match Input.query "UPN > " with
          | ""  → ()
          | "+" → stacks.Top.Push (stacks.Top.Pop + stacks.Top.Pop)
          | "*" → stacks.Top.Push (stacks.Top.Pop * stacks.Top.Pop)
          | "." → putline (show stacks.Top.Top)
          | "exit" | "halt" | "q" | "quit" | "stop" → raise EOF
          | "(" → stacks.Push (stack ())
          | ")" → stacks.Pop |> ignore
          | s   → stacks.Top.Push (read-nat s)
        with
          | Pop | Top → putline "stack is empty"
          | Read      → putline "enter a number or an operator"
    with
      | EOF → putline "bye bye"
```