Programmiersprachen

Abstraktion

Berthold Hoffmann

Studiengang Informatik
Universität Bremen

Sommersemester 2006
(Vorlesung am 29. Mai 2006)

Abstraktion (29. Mai 2006)

Vor-vorletztes Mal: Werte

Die funktionale Façette

- Datentypen definieren Wertemengen mit Operationen (Funktionen)
- Funktionen berechnen Werte aus Werten (und sind selber Werte!)
- Ausdrücke sind geschachtelte Funktionsanwendungen
- Die Auswertung eines Ausdrucks liefert einen Wert

$$eval_\varepsilon : \mathcal{A}_t \to \mathbb{W}_t \quad (t \in \mathcal{T}, \varepsilon : X \to \mathbb{W})$$

Vorletztes Mal: Speicher

Die imperative Façette

- Variablen benennen Speicherzellen
- Zellen können Variablen enthalten (Zeiger)
- Die Ausführung von Befehlen verändert den Zustand des Speichers

$$exec : \mathcal{C} \to (\mathcal{S} \to \mathcal{S}) \quad (\mathcal{S} : X \to \mathbb{W})$$

Letztes Mal: Bindung

Die deklarative Façette

- Vereinbarungen binden Programm-Größen an Bezeichner
- Blöcke erlauben das Schachteln von Vereinbarungen
- Die Abarbeitung von Vereinbarungen verändert den Programm-Kontext

$$elab : \mathcal{D} \to (\mathcal{C} \to \mathcal{C}) \quad (\mathcal{C} : X \to \mathcal{E})$$

Heute: Abstraktion

Erweiterung von Ausdrücken, Befehlen, Vereinbarungen um benutzerdefinierte Funktionen, Prozeduren, generische Vereinbarungen

- Funktionen, Prozeduren, generische Vereinbarungen
- Parameter und Übergabemechanismen
- Auswertungsreihenfolgen

Abstraktion (29. Mai 2006)

Die Idee

- wiederkehrende Progammstücke werden zusammengefasst und benannt
- Ihr Name abstrahiert von dem Programmstück, ihrem Rumpf
- Jedes Auftreten des Programmstücks wird durch einen Aufruf der Abstraktion ersetzt.
- Veränderliche Teile des Abstraktionsrumpfes werden zu formalen Parametern gemacht.
- Beim Aufruf werden aktuelle Parameter für die formalen eingesetzt
- Einsetzen des Rumpfes mit Substitution der Parameter

Weshalb heißt das Abstraktion?

Abstraktionen erlauben das Verbergen von Implementierungsdetails

- Benutzersicht
  - Was tut eine Abstraktion?
  - Vor- und Nachbedingung, "Effekt"
- Implementierersicht
  - Wie ist sie realisiert?
  - Algorithmus, Effizienz, ...

Intuitive Bedeutung einer Abstraktion

$$\begin{array}{l} p(x) :\Leftrightarrow B \\ \dots \\ p(A) \\ \dots \\ p(A') \\ \dots \end{array} \quad \Leftrightarrow \quad \begin{array}{l} p(x) :\Leftrightarrow B \\ \dots \\ B[x/A] \\ \dots \\ B[x/A'] \\ \dots \end{array}$$

- "$B[x/A]$" ersetzt alle freien Auftreten von x in B durch A
  Was bedeutet wohl "frei" in diesem Zusammenhang?
- mehrere Parameter entsprechen einem Produkt-Parameter
- Vorsicht: Rekursion?

Funktionsabstraktion

- Eine Funktionsabstraktion abstrahiert von einem Ausdruck
- Jeder Aufruf wird ausgewertet und liefert einen Wert

Beispiel (Potenzieren in ML)

```
fun power (x: real, n: int) =
    if n = 0 then 1 else x * power (x, n-1)
```

- Was? $\forall x \in \mathbb{W}_{\texttt{float}} \forall n \in \mathbb{W}_{\texttt{int}} : n \geqslant 0 \Rightarrow \texttt{power}(x, n) = x^n$
- Wie? rekursives Multiplizieren, Aufwand $\mathcal{O}(n)$

Eine andere Funktionsabstraktion

Beispiel ("Effizientes" Potenzieren in ML)

```
fun power (x: real, n: int) =
    if n = 0
    then 1
    else if even (n)
         then power (sqr (x), n div 2)
         else power (sqr (x), n div 2) * x
```

- Was? ... $\texttt{power}(x, n) = x^n$ (wie vorher)
- Wie? rekursives Multiplizieren der Quadrate, Aufwand $\mathcal{O}(\log n)$

Anonyme Funktionsabstraktion

Beispiel (power in ML)

```
fun power (x: real, n: int) =
    if n = 1 then 1 else x * power (x, n-1)
```

ist eine Abkürzung für die Wertvereinbarung

```
val power = fn (x: real, n: int) =¿
                if n = 1 then 1 else x * power (x, n-1)
```

Funktionabstraktionen müssen nicht benannt werden!

(**fn** (x: real) ⇒ x * x * x) 3 ⇝ 27

Besonders nützlich für Funktionsparameter:

... integral((0.0,0.1,**fn** (x: real) ⇒ x * x * x) ...

Funktionsabstraktion in Ada

- Eine Funktionsabstraktion abstrahiert von einem Ausdruck
- Jeder Aufruf liefert einen Wert

Beispiel (Potenzieren in Ada)

```
function power (x : float; n : Integer) return float is
        p: float := 1;
    begin (* Annahme: n ⩾ 0 *)
    for i in 1.. n loop p := x * p; end loop;
    return p;
    end power;
```

- Der Rumpf ist ein Befehl (hoffentlich mit **return**'s)
- Auch globale Variablen dürfen benutzt und verändert werden

Prozedurabstraktion

- Eine Prozedurabstraktion abstrahiert von einem Befehl
- Jeder Aufruf wird ausgeführt und verändert den Speicherzustand

Beispiel (Sortieren in Ada)

```
type WordSequence is array (...) of Word;
procedure sort (in out words : WordSequence);
...
```

- Was? `sort` sortiert ein Feld
- Wie? z. B. *insertion sort*
- Jeder Aufruf `sort(v)` verändert den Zustand von v

Abstraktionsprinzip

- Prozeduren abstrahieren von Befehlen
- Funktionen abstrahieren von Ausdrücken

Das kann man zum Prinzip erheben:

Abstraktionsprinzip Für alle Programmstücke, die Berechnungen verkörpern, können Abstraktionen gebildet werden.

Was käme da noch in Frage?

- Selektorabstraktionen abstrahieren von Variablenzugriffen
- Generische Vereinbarungen abstrahieren von Vereinbarungen

Selektorabstraktion

- Eine Selektorabstraktion abstrahiert von einem Variablenzugriff ("*lvalue*" in C++)
- Jeder Aufruf liefert eine Variable

Beispiel (Selektor (fiktiv))

```
type Queue is record item : array (1..max) of Integer;
                     front, rear, count : 0..max
                end record;
selector first (in q : Queue) : Integer is q.item[q.front];
... first (queueA) := first (queueA) - 1;
```

- Was? wählt das erste Schlangenelement aus
- Wie? durch Indizieren von `item`
- Andere Darstellung – anderer Zugriff

Generische Abstraktion

- Eine generische Abstraktion abstrahiert von einer Vereinbarung
- Jede Instantiierung liefert eine Vereinbarung

Dieses mächtige Konzept wird beim Thema Kapselung besprochen

Abstraktion (29. Mai 2006)

Parameterisierung

Erst Parameter machen Abstraktionen flexibel

Beispiel (Kreisumfang)

```
val pi = 3.1416;
fun circum (r: real) = 2 * pi * r

...circum(1.0) ...circum(a+b) ...
```

- Formaler Parameter (`r`): Bezeichner im Rumpf der Abstraktion
- Aktueller Parameter (`1.0`, `a+b`): Programmstück, das beim Aufruf angegeben wird, z. B. ein Ausdruck
- Argument: der Wert, der an die Abstraktion übergeben wird, z. B. der Wert von `a+b`

Argument-Arten

- Werte
- Variablen
- Prozedur- und Funktionsabstraktionen
- Typen
  ein mächtiges Konzept, behandelt erst unter Kapselung

Parameter-Modus

- Eingang (**in**): vom Aufrufer an die Abstraktion
- Ausgang (**out**): von der Abstraktion an den Aufrufer (u. a. Ergebniswert bei Funktionsabstraktionen)
- Durchgang (**in-out**): vom Aufrufer an die Abstraktion und zurück

Wie werden diese Modi realisiert?
Parameter-Übergabemechanismen

1. Kopieren
2. Alias-Definition

Kopier-Mechanismen

$f(A) \rightsquigarrow$ **fun** $f(x : t)$ **begin** B **end**

- Der formale Parameter x ist wie eine lokale Variable
- Der Austausch zwischen aktuellem Parameter A und x geschieht durch Zuweisung

Modus	Name	aktueller P.	Aktion bei Eintritt	Aktion bei Austritt
in	Wert-Par.	Ausdruck	$x := A$	—
out	Ergebnis-Par.	Variablenzugriff	—	$A := x$
in-out	Wert-Ergebnis-P.	Variablenzugriff	$x := A$	$A := x$

Beispiel 1: Kopier-Mechanismen

Beispiel (in und out)

```
type Vector is array (1..n) of float;

procedure add(in v, w : Vector; out sum : Vector) is
(1)   begin
        for i in 1 .. n
        loop sum(i) := v(i) + w(i); end loop;
(2)   end sum;
...   sum(a,b,c);
```

Aktionen zur Parameterübergabe:

(1) v : Vector := a; w : Vector := b; u : Vector;
(2) c := u;

Beispiel 2: Kopier-Mechanismen

Beispiel (in-out)

```
procedure normalize (in out u: Vector) is
(3)   s : float := 0.0;
      begin
      for i in 1 .. n loop s := s + sqr (u(i)); end loop;
      s := sqrt (s);
      for i in 1 .. n loop u(i) := u(i) / s; end loop;
(4)   end normalize;
...   normalize(c);
```

Aktionen zur Parameterübergabe:

(3) u : Vector := c;

(4) c := u;

Definierender Mechanismus

$$f(A) \rightsquigarrow \textbf{fun}\ f(x:t)\ \textbf{begin}\ B\ \textbf{end}$$

- In B ist der formale Parameter x ist ein **Alias** für den aktuellen Parameter A
- Jede Zugriff auf x in B greift indirekt auf A zu
- So können alle Parameter-Arten übergeben werden:
  - Eine Konstante x bezeichnet den Wert des Ausdrucks A
  - Eine Variable x ist ein Alias für den Variablenzugriff A
  - Eine (Funktions-) Prozedur x ist ein Name für die Abstraktion A
  - (Ein Typ x ist ein Synonym für den Typausdruck A)
- Variablen-Parameter können jeden Modus realisieren (**in**, **out**, **in-out**)

Beispiel 1: Definierender Mechanismus

Beispiel (in und out)

```
type Vector is array (1..n) of float;
procedure add(in v, w : Vector; out sum : Vector) is
(1)   begin
      for i in 1 .. n
      loop sum(i) := v(i) + w(i); end loop;
(2)   end sum;
...   sum(a,b,c);
```

Aktionen zur Parameterübergabe:

(1) v **renames** a; w **renames** b; u **renames** c
Jeder Zugriff auf u, v, w greift indirekt auf a, b bzw. c zu

(2) —

Beispiel 2: Definierender Mechanismus

Beispiel (in-out)

```
procedure normalize (in out u: Vector) is
(3)   s : float := 0.0;
      begin
      for i in 1 .. n loop s := s + sqr (u(i)); end loop;
      s := sqrt (s);
      for i in 1 .. n loop u(i) := u(i) / s; end loop;
(4)   end normalize;
...   normalize(c);
```

Aktionen zur Parameterübergabe:

(3) u **renames** c

(4) —

Problem: Alias

Beispiel (Verwirrung)

```
procedure confuse (var m, n : Integer) is
  begin
  n := 1; n := m + n
  end confuse;
  confuse(i,i);
```

- `i` bekommt immer den Wert 2!
- Aliase sind schwer zu entdecken!
  z. B. `confuse(a(i),a(j))`

Vergleich Kopierende / Definierende Mechanismen

$$f(A) \rightsquigarrow \textbf{fun}\ f(x:t)\ \textbf{begin}\ B\ \textbf{end}$$

Aspekt	Mechanismus: Kopieren	Mechanismus: Definieren
Modi	alle	alle
Arten	zuweisbare Werte	alle
Übergabe-Aufwand	$size(t)$	1
Zugriffe auf aktuellen P.	1-2	$use(x, B)$
Vorteile	"synchrone" Ü.	$\mathcal{O}(c)$
Nachteile	$\mathcal{O}(size(t))$:= vorausges. für t	"asynchrone" Ü. Aliase

Das Korrespondenzprinzip

Paramterspezifikationen ähneln Vereinbarungen

Das kann man zum Prinzip erheben

Korrespondenzprinzip Jeder Parameterspezifikation entspricht eine Vereinbarung – und umgekehrt.

Vereinbarung	Parameterspezifikation
Variablenvereinbarung	Wert-Parameter
Konstantenvereinbarung	Konstanten-Parameter
Variablen-Umbenennung	Variablen-Parameter
Prozedurvereinbarung	Prozedur-Parameter

Abstraktion (29. Mai 2006)

Auswertung der aktuellen Parameter

Wann sollen aktuelle Parameter ausgewertet werden?

Beispiel (Quadrat-Funktion)

let val a = 6 **val** b = 7
in fun sqr (n: int) = n * n
. . . sqr(a+b)

- Strikt (*call-by-value*, *innermost*)
  Erst Parameter auswerten, dann Funktion aufrufen
  $sqr(a+b) \rightsquigarrow sqr(42) \rightsquigarrow 1764$
- Normalisierend (*call-by-name*, *outermost*)
  Funktion aufrufen, bei Bedarf Parameter auswerten
  $sqr(a+b) \rightsquigarrow (a+b)*(a+b) \rightsquigarrow 42*42 \rightsquigarrow 1764$
- Verzögert (*call-by-need*, *lazy*) wie normalisierend, nach erster Benutzung Wert wiederverwenden

Auswertungsreihenfolge und Termination

Beispiel (Fakultät)

val a = 1
fun fac (n: int) = **if** n=0 **then** 1 **else** fac(n-1)
fac(a+1)

- Strikte Auswertung terminiert nicht
  $\mathrm{fac}(a1) \rightsquigarrow \mathrm{fac}(2) \rightsquigarrow \mathrm{fac}(1) \rightsquigarrow \mathrm{fac}(0) \rightsquigarrow \mathrm{fac}(-1) \rightsquigarrow \cdots$
- Normalisierende Auswertung terminiert
  $\mathrm{fac}(a+1)$
  $\rightsquigarrow$ **if** $(a+1)=0$ **then**1 **else***fac*$(a+1-1) \rightsquigarrow fac(a+1)-1)$
  $\rightsquigarrow$ **if** $(a+1-1)=0$ **then** 1 **else** $fac((a+1-1)-1) \rightsquigarrow 1$
- Verzögerte Auswertung auch

Abstraktion (29. Mai 2006)

Zusammenfassung

- Abstraktion strukturiert Algorithmen (im Kleinen)
- Parameter machen Abstraktionen flexibel
- Bei Abstraktionsanwendungen gibt es verschiedene Auswertungsreihenfolgen

Nächstes Mal: Ablaufsteuerung

Erst am 12. Juni! (Am 5. Juni ist Pfingstmontag)

- Sprünge (**goto**)
- Auswege (**break**, **continue**, **return** . . .)
- Ausnahmen (*exception*)

Zusammenfassung der bisher behandelten

- Konzepte
- Eigenschaften
- Prinzipien