Abstraktion

Markus Roggenbach

18. November 2002

Vorbemerkungen

Facetten von Programmiersprachen

- Werte – Ausdrücke
- Zustände – Befehl
- Bindungen – Vereinbarung

Idee der Abstraktion

Parametrisieren

- von Ausdrücken $\leadsto$ Funktionsabstraktion

Idee der Abstraktion

Parametrisieren

- von Ausdrücken $\leadsto$ Funktionsabstraktion
- von Befehlen $\leadsto$ Prozedurabstraktion

Idee der Abstraktion

Parametrisieren

- von Ausdrücken $\leadsto$ Funktionsabstraktion
- von Befehlen $\leadsto$ Prozedurabstraktion
- von Vereinbarungen $\leadsto$ generische Abstraktion

Motivationen

„Spracherweiterung“

Motivationen

„Spracherweiterung“
Strukturierungsprinzip (im Kleinen)

Motivationen

„Spracherweiterung“

Strukturierungsprinzip (im Kleinen)

Differenzierung: „Effekt“ versus Implementierung

Abstraktionsarten I: Funktionsabstraktion

Eine Funktionsabstraktion abstrahiert von einem Ausdruck.

```
function F (FP_1; ...; FP_n): T = B
```

Rumpf: Ausdruck.

Funktionsaufruf (liefert Wert): Ausdruck.

Beispiel

```
function power (x:Real;n:Integer):Real;
begin
  if n=1 then power:=x
         else power:=x*power(x,n-1)
end
```

Eigenschaften von Funktionsabstraktionen

Eine Funktionsabstraktion heißt

seiteneffektfrei, wenn sie den globalen Zustand eines Programms nicht verändert.

Eigenschaften von Funktionsabstraktionen

Eine Funktionsabstraktion heißt

seiteneffektfrei, wenn sie den globalen Zustand eines Programms nicht verändert.

referentiell transparent, wenn ihr Wert (bei gleichen Parameterwerten) unabhängig vom Kontext ihres Aufrufs ist.

Ein Beispiel

Pascal	Haskell
`var x: Char;`	`x = newIORef::IO(IORef Char)`

`function hugo (y:Int): Int;`
`begin`
`  x:='c';`
`  hugo:= 2 * y`
`end;`
	`hugo:: Int -> IOInt`
	`hugo (y::Int) =`
	
	`  writeIORef x 'c' >>`
	`  return 2 * y`

	`erna :: Int -> Int` `  erna (y::Int) = 2 * y`

Standardfunktion auf IORef:

```
>> :: IO alpha -> IO beta -> IO beta
```

Abstraktionsarten II: Prozedurabstraktion

Eine Prozedurabstraktion abstrahiert von einem Befehl.

```
procedure P (FP_1; ... ; FP_n) = B
```

Rumpf: Befehl

Prozeduraufruf (verändert Zustand): Befehl.

Beispiel in Ada

```
procedure swap (a: in out integer,
                b: in out integer) is
  temp : integer
  begin
    temp:=a;
    a := b;
    b := temp
  end swap
```

Abstraktionsarten III: Generische Abstraktion

Eine generische Abstraktion abstrahiert von einer Vereinbarung.

```
generic FP_1; ... ; FP_n package P is B
```

Rumpf: Vereinbarung.
generische Instantiierung (liefert Bindung): Vereinbarung.

Beispiel in Ada: Deklaration

```
generic capacity: in Positive;
package queue_class is
  procedure append (newitem: in Character);
  procedure remove (olditem: out Character);
end queue_class;
package body queue_class is
  items: array (1..capacity) of Character;
  ...
end queue_class;

package line_buffer is new queue_class (120);
```

Abstraktion – allgemein

abstrahieren = „absehen von etwas“

Abstraktion – allgemein

abstrahieren = „absehen von etwas“

in Programmiersprachen:

1. (parametrisiertes) Zusammenfassen von Elementen einer Facette zu einem neuen Element dieser Facette.
2. Binden der Abstraktion (an einen Namen).

Abstraktionsprinzip

Über jeder syntaktischen Klasse
können Abstraktionen konstruiert werden.

Abstraktionsprinzip

Über jeder syntaktischen Klasse
können Abstraktionen konstruiert werden.

(sofern diese Klasse „Berechnungen“ durchführt.)

Abstraktionsprinzip

Über jeder syntaktischen Klasse
können Abstraktionen konstruiert werden.

(sofern diese Klasse „Berechnungen“ durchführt.)

Bsp.: Variablenzugriffe

Parametermechanismen

Vereinbarung :	`function F (FP_1;...;FP_n) T = B`
Aufruf:	`F (AP_1;...;AP_n)`
Vereinbarung :	`procedure P (FP_1;...;FP_n) = B`
Aufruf:	`P (AP_1;...;AP_n)`
Vereinbarung :	`generic FP_1;...;FP_n package P is B`
Aufruf:	`package X is new P(AP_1;...;AP_n)`

`FP_i:` formale Parameter

`AP_i:` aktuelle Parameter

Zuordnung: aktuell $\leadsto$ formal

Positionszuordnung: `AP_i` wird `FP_i` zugeordnet.

Zuordnung: aktuell $\rightsquigarrow$ formal

Positionszuordnung: `AP_i` wird `FP_i` zugeordnet.

Namentliche Zuordnung: Name des formalen Parameters.

Zuordnung: aktuell $\rightsquigarrow$ formal

Positionszuordnung: `AP_i` wird `FP_i` zugeordnet.

Namentliche Zuordnung: Name des formalen Parameters.

```
function ComputePay
  (Income: Flaot; TaxRate: Float) return Float;

pay := ComputePay
   (TaxRate => 0.15, Income => 20000.0)
```

Modus versus Implementierung

Modus versus Implementierung

Modus: in

out

in-out

Modus versus Implementierung

Modus: in
out
in-out

Implementierung: kopierender Mechanismus
definierender Mechanismus

Kopierender Mechanismus

Zuweisungen zwischen AP und FP.

Kopierender Mechanismus

Zuweisungen zwischen AP und FP.

Modus	Aktueller Parameter	Eintritt	Austritt
in	Ausdruck	FP:=AP	–
out	Variable	–	AP:=FP
in-out	Variable	FP:=AP	AP:=FP

FP bezeichnet eine *lokale Variable* der Abstraktion.

Definierender Mechanismus

Bindung des FPs an den AP.

Definierender Mechanismus

Bindung des FPs an den AP.

Konstantenparameter: Wert

Definierender Mechanismus

Bindung des FPs an den AP.

Konstantenparameter: Wert

Variablenparameter: Verweis auf eine Variable

Definierender Mechanismus

Bindung des FPs an den AP.

Konstantenparameter: Wert

Variablenparameter: Verweis auf eine Variable

Prozedurparameter: Prozedurabstraktion

Definierender Mechanismus

Bindung des FPs an den AP.

Konstantenparameter: Wert

Variablenparameter: Verweis auf eine Variable

Prozedurparameter: Prozedurabstraktion

Funktionsparameter: Funktionsabstraktion

Korrespondenzprinzip

Zu jeder Form von Vereinbarung
gibt es eine korrespondierende Form
der Parameterübergabe.

Vorsicht bei out/in-out Parametern

```
procedure bloed (out x, out y)
  x:=7;
  y:=9
end

bloed(p,p)
```

Alias-Probleme bei Variablenparametern

direktes Problem:

```
void fun(int *first, int *second)
fun (&total, &total)
```

Alias-Probleme bei Variablenparametern

direktes Problem:

```
void fun(int *first, int *second)
fun (&total, &total)
```

unentscheidbares Problem:

```
void fun(int *first, int *second)
fun (&list[i], &list[j])
```

Formaler Parameter und nicht-lokale Variable:

```
var global: integer;

procedure hugo(var local: integer);
begin
  ...
end;

begin
  hugo(global);
  ...
end
```

Kopier- versus Definierender Mechanismus

Kriterium	Kopierm.	def. M.
zus. Speicherbedarf	x	
Zeit zum Kopieren	x	
Zugriff	„schnell“	„langsam“
Typen	mit Zuweisung	alle
„Sicherheit“ im nebenläufigen Kontext	ja	nein

Parameter Mechanismen in Programmiersprachen

Sprache	in call by value	out	in-out kopierend	in-out definierend call by reference
C	x			(pointer)
Pascal	x			(var)
Modula-2	x			(var)
Java	x			(object param.)

Fortran: in-out Modus

Typüberprüfung

Sprachen mit Typüberprüfung:

Fortran 77, Pascal, Modula-2, Ada, Java, Fortran 90

Typüberprüfung

Sprachen mit Typüberprüfung:

Fortran 77, Pascal, Modula-2, Ada, Java, Fortran 90

Sprachen ohne Typüberprüfung:

C im Original

ANSI-C: wahlweise Typüberprüfung.

Prinzip der Typvollständigkeit

Keine „Operation“
sollte in den Typen ihrer Operanden
eingeschränkt sein.

Prinzip der Typvollständigkeit

Keine „Operation“
sollte in den Typen ihrer Operanden
eingeschränkt sein.

Ergebnisse von Funktionsabstaktionen:

Prinzip der Typvollständigkeit

Keine „Operation“
sollte in den Typen ihrer Operanden
eingeschränkt sein.

Ergebnisse von Funktionsabstaktionen:

Fortran 77, Pascal, Modula-2: einfache Typen.

Prinzip der Typvollständigkeit

Keine „Operation“
sollte in den Typen ihrer Operanden
eingeschränkt sein.

Ergebnisse von Funktionsabstaktionen:

Fortran 77, Pascal, Modula-2: einfache Typen.

C: alle Typen bis auf „array“ und „function“.

Prinzip der Typvollständigkeit

Keine „Operation“
sollte in den Typen ihrer Operanden
eingeschränkt sein.

Ergebnisse von Funktionsabstaktionen:

Fortran 77, Pascal, Modula-2: einfache Typen.

C: alle Typen bis auf „array“ und „function“.

Ada: einzige imperative Sprache, die „alle“ Typen erlaubt.

Auswertungsreihenfolge

strikt (applikativ, eager):

1) Auswertung bei Aufruf der Abstraktion $\leadsto w$
2) Bindung: $FP \mapsto w$.

Auswertungsreihenfolge

strikt (applikativ, eager):

1) Auswertung bei Aufruf der Abstraktion $\leadsto w$
2) Bindung: $FP \mapsto w$.

normalisierend (normal order):

1) Bindung: $FP \mapsto AP$.
2) Auswertung bei jeder Verwendung von FP.

faul (lazy):

1) Bindung: $FP \mapsto AP$.

2) Auswertung bei der ersten Verwendung von $FP : \rightsquigarrow w$

3) Bindung: $FP \mapsto w$.

Beispiele

```
fun sqr (n: int) = n * n

sqr(p+q)

fun cand (b1:bool, b2:bool) =
  if b1 then b2 else false

cand( n > 0, t/0 > 1)
```