

Programmiersprachen
Vorlesung 7 vom 20.11.23
Prozeduren und Funktionen

Christoph Lüth

Universität Bremen

Wintersemester 2023/24

Organisatorisches

► Vorträge

Liste möglicher Sprachen

- ▶ Logische Programmierung: Prolog, Oz
- ▶ Nebenläufig/Reaktiv: Erlang, Golang
- ▶ Abhängige Typen: Idris, Agda, Liquid X (Dependent types)
- ▶ Prozedural: Julia, Kotlin, Swift
- ▶ Skriptsprachen: Lua, Tcl
- ▶ Funktional: SML/OCAML, Elm, Clojure/Lisp, Scala, Elixir
- ▶ Stack-basiert: Forth
- ▶ Historisch: COBOL, Algol-68, APL, Ada, ABAP, Smalltalk
- ▶ Datenflusssprachen: Id, Lucid, Lustre
- ▶ Shiny: Mojo, Dart
- ▶ DSLs: R, SQL, Postscript, TeX, Verilog/VHDL, SystemC, SpinalHDL

Wo sind wir?

- ▶ Einführung
- ▶ Einfache Ausdrücke und Werte
- ▶ Einfache Typen, Anweisungen und Seiteneffekte
- ▶ Variablen und Speichermodelle
- ▶ Aggregierende Typen
- ▶ Ausnahmen und Fehlerbehandlung
- ▶ **Prozeduren und Funktionen**
- ▶ Fortgeschrittene Typsysteme
- ▶ Datenabstraktion
- ▶ Programmierparadigmen
- ▶ Skriptsprachen
- ▶ Ab Woche 11 (2024): Studentische Vorträge.

Prozeduren und Funktionen

- ▶ **Prozeduren** sind benannte, parameterisierte **Blöcke**
 - ▶ Meist ohne Rückgabewert
- ▶ **Funktionen** sind Prozeduren mit Rückgabewert
 - ▶ **Reine** Funktionen (pure functions): referentiell transparent, ohne Seiteneffekt
 - ▶ Meist **mit** Seiteneffekten
- ▶ Viele Programmiersprachen unterscheiden das nicht
- ▶ Funktionsdefinition hat **(formale) Parameter**, beim Aufruf **Parameterwerte** (Argumente)

```
int f(x) { return x*10; }    // 'x' ist formaler Parameter
```

```
... f(29+2) ...    // '29+2' ist Parameterwert
```

Parameterübergabe (Parameter Passing)

- ▶ Aus **konzeptioneller** Sicht gibt es drei Arten von Parametern:
 - ▶ **Eingabeparameter** — erlaubt Kommunikation vom Aufrufer an die Funktion
 - ▶ **Ausgabeparameter** — erlaubt Kommunikation von der Funktion an den Aufrufer
 - ▶ **Ein/Ausgabeparameter** — erlaubt bidirektionale Kommunikation

Parameterübergabe (Parameter Passing)

- ▶ Aus **konzeptioneller** Sicht gibt es drei Arten von Parametern:
 - ▶ **Eingabeparameter** — erlaubt Kommunikation vom Aufrufer an die Funktion
 - ▶ **Ausgabeparameter** — erlaubt Kommunikation von der Funktion an den Aufrufer
 - ▶ **Ein/Ausgabeparameter** — erlaubt bidirektionale Kommunikation
- ▶ Beispiel (Ada; Eingabeparameter `v`, `w`, Ausgabeparameter `sum`)

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

Parameterübergabe (Parameter Passing)

- ▶ Aus **konzeptioneller** Sicht gibt es drei Arten von Parametern:
 - ▶ **Eingabeparameter** — erlaubt Kommunikation vom Aufrufer an die Funktion
 - ▶ **Ausgabeparameter** — erlaubt Kommunikation von der Funktion an den Aufrufer
 - ▶ **Ein/Ausgabeparameter** — erlaubt bidirektionale Kommunikation
- ▶ Beispiel (Ada; Eingabeparameter `v`, `w`, Ausgabeparameter `sum`)

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

- ▶ Aus **operationaler** Sicht gibt verschiedene Arten der **Parameterübergabe**

Call by Value

- ▶ **Parameterwert** ist **beliebiger** Ausdruck (R-Wert)
- ▶ **Funktionsaufruf:**
 - ▶ Parameter wird zu v ausgewertet
 - ▶ Formaler Parameter wird lokale Variable im Funktionsrumpf, mit v initialisiert
 - ▶ Funktionsrumpf wird ausgeführt
- ▶ Für **Eingabeparameter**
- ▶ Klare Semantik (kein Effekt auf Aufrufer)
- ▶ Effizient für “kleine”, ineffizient für “große” Datenstrukturen (Felder etc.)
- ▶ Wertet eventuell zu viel aus

Call by Reference (Call by Variable)

- ▶ Parameterwert muss ein L-Wert sein
- ▶ Funktionsaufruf:
 - ▶ Umgebung des Funktionsrumpfes wird erweitert
 - ▶ Formaler Parameter wird zu L-Wert aufgelöst (aliasing)
 - ▶ Funktionsrumpf wird ausgeführt
- ▶ Für **Ausgabeparameter** und **Ein/Ausgabeparameter**
- ▶ Funktion kann Parameterwert verändern
- ▶ Effizient aber fehleranfällig (wegen Aliasing)

```
void foo(reference int x)
{ x= x+1; }

char V[10];
i= 2;
V[2]= 5;
foo(V[i]);
// V[2] == 6
```

Call By Name

- ▶ Parameterwert ist beliebiger Ausdruck e (R-Wert)
- ▶ Aufruf von Funktion $f(x)$ ist semantisch äquivalent zur Ausführung des Rumpfes, in dem alle x durch e ersetzt werden.
- ▶ Semantisch sauber, aber subtil
- ▶ Stammt von ALGOL-60, wird heute nur noch wenig benutzt

```
int x= 0;
int foo(name int y)
{
    int x= 2;
    return x+y;
}

...
int a= foo(x+1);
// = int x= 2; x+ x+ 1 ???
```

Jensen's Device

- ▶ Call-by-Name erlaubt **Metaprogrammierung** (Macros)
- ▶ Beispiel: Jensen's Device

```
int sum(name int exp; name int i; int fr; int to)
{
    int acc= 0;
    for (i= fr; i<= to; i++) acc= acc+ exp;
    return acc;
}
int x= ...;
int y= sum(2*x*x- 1, x, 1, 10)
```

- ▶ Berechnet

$$y = \sum_{x=1}^{10} 2x^2 - 1$$

Variationen

- ▶ Call by **constant**:
 - ▶ Wenn Funktionsrumpf den formalen Parameter nicht modifiziert kann call-by-value durch call-by-reference implementiert werden.
- ▶ Call by **need** (Haskell):
 - ▶ Ähnlich call-by-name, Parameterwert wird **nur** ausgewertet, wenn er benutzt wird
- ▶ Call by **value** mit Zeigern (C, Java, Python, Rust)
 - ▶ Wenn Werte Zeiger (Referenzen) sind kann der Aufruf Seiteneffekte haben
 - ▶ Parameter vom Typ Pointer (C) oder **Object** (Java, Python) sind Ein/Ausgabe-Parameter

Parameterübergabe in C

C-Standard (C99, §6.5.2.2)

1. The expression that denotes the called function⁷⁷ shall have type pointer to function returning void or returning an object type other than an array type.
4. An argument may be an expression of any object type. In preparing for the call to a function, the arguments are evaluated, and each parameter is assigned the value of the corresponding argument.^a

^aA function may change the values of its parameters, but these changes cannot affect the values of the arguments. On the other hand, it is possible to pass a pointer to an object, and the function may change the value of the object pointed to. A parameter declared to have array or function type is adjusted to have a pointer type as described in 6.9.1.

Parameterübergabe in Rust

- ▶ Ähnlich C/Java (call by value), aber **restriktiver: Ownership**
- ▶ Datenstrukturen haben einen **Owner**.
- ▶ Parameterübergabe:
 - ▶ Entweder als **unveränderliche Referenz**,
 - ▶ oder als **veränderliche Referenz**.
 - ▶ Muss in der Funktionsdefinition deklariert werden.

Falsch:

```
fn main() {  
    let s = String::from("hello");  
  
    let len= calc_len(&s); // Works  
    change(&s); // Does not work  
}  
  
fn change(some_string: &String) {  
    some_string.push_str(", world")  
}  
  
fn calc_len(str: &String)-> usize {  
    s.len()  
}
```

Paramterübergabe in Rust

- ▶ Ähnlich C/Java (call by value), aber **restriktiver: Ownership**
- ▶ Datenstrukturen haben einen **Owner**.
- ▶ Parameterübergabe:
 - ▶ Entweder als **unveränderliche Referenz**,
 - ▶ oder als **veränderliche Referenz**.
 - ▶ Muss in der Funktionsdefinition deklariert werden.
- ▶ Vergleiche **const** in C.

Richtig:

```
fn main() {  
    let mut s = String::from("hello");  
  
    let len= calc_len(&s); // Works  
    change(&mut s); // Works  
}  
  
fn change(some_string: &mut String) {  
    some_string.push_str(", world")  
}  
  
fn calc_len(str: &String)-> usize {  
    s.len()  
}
```


Parameterübergabe und Auswertungsstrategie

- ▶ Auswertungsstrategien in funktionalen Sprachen:
 - ▶ Innermost-first
 - ▶ Outermost-first
- ▶ Innermost-first $\sim$ call-by-value, eager evaluation
- ▶ Outermost-first $\sim$ call-by-need, lazy evaluation
- ▶ Outermost-first: nicht-strikt

```
f 7 undefined  $\rightsquigarrow$  14
```

Beispiel:

```
f x y = x + x
```

Auswertung innermost:

```
f (f 7 3) (f 5 9)
 $\rightsquigarrow$  f (7+7) (5+ 5)
 $\rightsquigarrow$  f 14 20
 $\rightsquigarrow$  14+14  $\rightsquigarrow$  28
```

Auswertung outermost:

```
f (f 7 3) (f 5 9)
 $\rightsquigarrow$  f 7 3+ f 7 3
 $\rightsquigarrow$  (7+ 7)+ (7+ 7)
 $\rightsquigarrow$  14+14  $\rightsquigarrow$  28
```

Funktionen höherer Ordnung

Funktionen Höherer Ordnung

- ▶ Funktionen höherer Ordnung sind Funktionen $A \rightarrow B$ mit A oder B eine Funktion.
- ▶ Funktion als Argument, Beispiel (Python):

```
map(str, [1, 18, true, "foo"])
```

- ▶ Funktion als Resultat, Beispiel $3 \leq$ (vom Typ `Int` $\rightarrow$ `Bool`) in (Haskell):

```
filter (3 <=) [0,7,1,8,2,9,-2]
```

- ▶ Dabei hilfreich: “anonyme” Funktionen (Lambda-Ausdrücke), Beispiel (Python):

```
filter (lambda x: 3<= x, [0,7,1,8,2,9,-2])
```

- ▶ Komplikationen: Scoping
- ▶ Python und besonders Haskell unterstützen Funktionen höher Ordnung

Funktionen höherer Ordnung in C

- ▶ Auch C unterstützt Funktionen höherer Ordnung — durch Zeiger
- ▶ Beispiel:

```
typedef struct list_t {  
    void          *elem;  
    struct list_t *next;  
} list_t;  
extern list_t *filter(int f(void *x), list_t *l);  
extern list_t *map(void *f(void *x), list_t *l);
```

- ▶ Problem: Speicherverwaltung, Typsystem nicht expressiv genug
- ▶ Wird genutzt für Sprungtabellen, Signalhandler, Callbacks.

Dynamische Bindung

- ▶ Methode `f` einer Klasse kann auf allen Untertypen angewandt werden.
- ▶ Konkrete Klasse der Instanz bestimmt konkrete Methode (**dynamische Bindung**)

```
class C:
    def f(self):
        print("Foo.")
    def g(self):
        print("Baz.")

class D(C):
    def f(self):
        print("Wibble.")
```

Semantik

Funktionsparameter

```
int foo(int x)
{
    x= 2*x +1;
    return x;
}
```

- ▶ Was ist der Parameter `x`?
- ▶ Lokale Variable mit unbestimmten (aber definierten) initialen Wert.
- ▶ Initialer Wert wird bei Funktionsaufruf **übergeben**.

Funktionsparameter

```
int foo(int x)
{
    x= 2*x +1;
    return x;
}
```

- ▶ Was ist der Parameter `x`?
- ▶ Lokale Variable mit unbestimmten (aber definierten) initialen Wert.
- ▶ Initialer Wert wird bei Funktionsaufruf **übergeben**.
- ▶ Benötigen semantischen Mechanismus, der diese **Übergabe** modelliert.

Ein Programm

$$\begin{aligned}\phi \equiv & \textbf{fun } f_1(x_{1,1}, \dots, x_{1,n_1}) = b_1 \\ & \textbf{fun } f_2(x_{2,1}, \dots, x_{2,n_2}) = b_2 \\ & \dots \\ & \textbf{fun } f_m(x_{m,1}, \dots, x_{m,n_m}) = b_m\end{aligned}$$

- ▶ Programme haben an sich keine Semantik.
- ▶ Stattdessen: jede Funktion hat eine Semantik.

Parameterized Blocks

- ▶ Block mit **Parameter**

$$pb ::= \lambda i.pb \mid c$$

- ▶ Damit:

$$\begin{aligned}\phi \equiv & \text{fun } f_1 = \lambda x_{1,1}, \dots, x_{1,n_1}. b_1 \\ & \text{fun } f_2 = \lambda x_{2,1}, \dots, x_{2,n_2}. b_2 \\ & \dots \\ & \text{fun } f_m = \lambda x_{m,1}, \dots, x_{m,n_m}. b_m\end{aligned}$$

- ▶ Müssen Umgebung Γ mit **Funktionen** erweitern: $\Gamma_{\text{fun}}(\phi) = \{(f_i, pb_i) \mid (f_i, pb_i) \in \phi\}$

Funktionsaufruf

Modelliert durch **Substitution**:

$$n[e/x] = n$$

$$y[e/x] = \begin{cases} e & x = y \\ y & \text{otherwise} \end{cases}$$

$$(e_1 + e_2)[e/x] = (e_1[e/x]) + (e_2[e/x])$$

...

$$(c_1; c_2)[e/x] = (c_1[e/x]); (c_2[e/x])$$

$$(\text{if } (b) \text{ then } c_1 \text{ else } c_2)[e/x] = \text{if } (b[e/x]) \text{ then } c_1[e/x] \text{ else } c_2[e/x]$$

$$(\text{while } (b) \text{ c})[e/x] = \text{while } (b[e/x]) \text{ (c[e/x])}$$

$$(\lambda y. c)[e/x] = \begin{cases} \lambda y. c & x = y \\ \lambda y. (c[e/x]) & x \neq y, y \notin \text{FV}(e) \\ \lambda z. ((c[z/y])[e/x]) & x \neq y, y \in \text{FV}(e), z \notin \text{FV}(e) \cup \text{FV}(c) \end{cases}$$

Extending the Language

$$\begin{aligned} l &::= i \mid l.i \mid l[e] \\ e &::= \mathbb{Z} \mid \text{true} \mid \text{false} \mid l \\ &\quad \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 / e_2 \\ &\quad \mid e_1 == e_2 \mid e_1 < e_2 \\ &\quad \mid !e \mid e_1 \&\& e_2 \mid e_1 || e_2 \\ &\quad \mid l := e \mid \text{throw}(x) \\ &\quad \mid f(e_1, \dots, e_n) \\ c &::= e \mid \text{if } (e) \text{ then } c_1 \text{ else } c_2 \mid \text{while } (e) \text{ } c \mid c_1; c_2 \mid \text{nil} \mid \text{try } c_1 \text{ catch } x \rightarrow c_2 \\ &\quad \mid \text{return } e \\ pb &::= \lambda i. pb \mid c \end{aligned}$$

Funktionsaufruf

► Werte zurückgeben: $E = \{E_R\} \times \mathbf{V} \cup \{E_0, E_1\}$

► Damit:

$$\frac{\Gamma(f) = \lambda x. b \quad \langle b[e/x], \sigma \rangle \rightarrow_{Stmt} \langle (E_R, v), \sigma' \rangle \quad v \in \mathbf{V}}{\Gamma \vdash \langle f(e), \sigma \rangle \rightarrow_{Exp} \langle v, \sigma' \rangle} \quad (1)$$

Funktionsaufruf

► Werte zurückgeben: $E = \{E_R\} \times \mathbf{V} \cup \{E_0, E_1\}$

► Damit:

$$\frac{\Gamma(f) = \lambda x. b \quad \langle b[e/x], \sigma \rangle \rightarrow_{Stmt} \langle (E_R, v), \sigma' \rangle \quad v \in \mathbf{V}}{\Gamma \vdash \langle f(e), \sigma \rangle \rightarrow_{Exp} \langle v, \sigma' \rangle} \quad (1)$$

$$\frac{\Gamma \vdash \langle e, \sigma \rangle \rightarrow_{Exp} \langle v_1, \sigma' \rangle \quad \Gamma(f) = \lambda x. b \quad \langle b[v_1/x], \sigma' \rangle \rightarrow_{Stmt} \langle (E_R, v_2), \sigma' \rangle \quad v_1, v_2 \in \mathbf{V}}{\Gamma \vdash \langle f(e), \sigma \rangle \rightarrow_{Exp} \langle v_2, \sigma'' \rangle} \quad (2)$$

Was fehlt hier?

- ▶ Mehrere Argumente
- ▶ Exceptions im Funktionsrumpf
- ▶ “Fall-through”: kein `return` am Ende