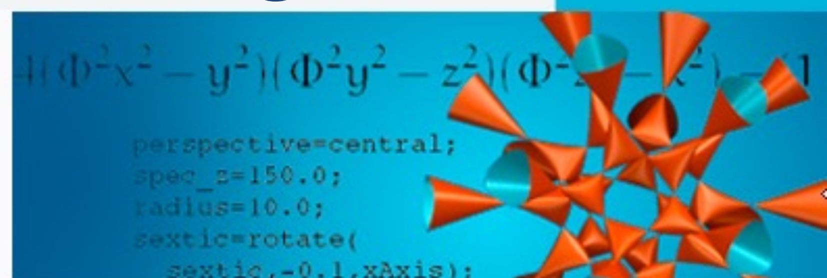
Declarative programming

Summer semester 2024

Prof. Christoph Bockisch, Steffen Dick
(Programming languages and tools)

Imke Gürtler, Daniel Hinkelmann, Aaron Schafberg, Stefan Störmer



[Script 11]

Abstraction

- Avoidance of redundancy (Don't Repeat Yourself)
- Better readability
- Known abstraction mechanisms
  - Constants
  - Functional abstraction

Functional abstraction

```
(list-of String) -> Boolean
; does l contain "dog"
(define (contains-dog? l)
  (cond
    [(empty? l) false]
    [else
      (or
        (string=? (first l) "dog")
        (contains-dog?
          (rest l))))]))
```

```
(list-of String) -> Boolean
; does l contain "cat"
(define (contains-cat? l)
  (cond
    [(empty? l) false]
    [else
      (or
        (string=? (first l) "cat")
        (contains-cat?
          (res
```

The only differences

Functional abstraction

```
; String (list-of String) -> Boolean
; to determine whether l contains the string s
(define (contains? s l)
  (cond
    [(empty? l) false]
    [else ( or (string=? (first l) s)
      (contains? s (rest l)))]))
```

Difference as a parameter for a function. Reuse of the commonality.

```
(list-of String) -> Boolean
; does l contain "dog"
(define (contains-dog? l)
  (contains? "dog" l))

(list-of String) -> Boolean
; does l contain "cat"
(define (contains-cat? l)
  (contains? "cat" l))
```

Functional abstraction

```
; (list-of Number) -> Number
; adds all numbers in l
(define (add-numbers l)
  (cond
    [(empty? l) 0]
    [else
      (+ (first l)
        (add-numbers (rest l))))))

; (list-of Number) -> Number
; multiplies all numbers in l
(define (mult-numb
  (cond
    [(empty? l) 1]
    [else
      (* (first l)
        (mult-num
```

Different values can be encapsulated as parameters.

What about different function calls?

Solution idea

```
(define (op-numbers op z l)
  (cond
    [(empty? l) z]
    [else
      (op (first l)
        (op-numbers op z (rest l)))]))
(define (add-numbers l) (op-numbers + 0 l))
(define (mult-numbers l) (op-numbers * 1 l))
> (add-numbers (list 1 2 3 4))
```

Coding the function to be called as a parameter. Is that possible?

function call: expected a function after the open parenthesis, but found a variable

Limitation of BSL (Beginning Student Laguage)

Functional abstraction

- Solution approach
  - Coding of function names as parameters in function headers
  - Passing function names as arguments when calling functions
  - Use of parameters instead of function names in function body

- Why does the solution approach fail?
  - Previously: Functions are not values

- Solution: Treat functions like values!
  - Switching the language level to "Intermediate level with lambda"

Functions as parameters

```
(define (op-numbers op z l)
  (cond
    [(empty? l) z]
    [else
      (op (first l)
        (op-numbers op z (rest l)))]))
(define (add-numbers l) (op-numbers + 0 l))
(define (mult-numbers l) (op-numbers * 1 l))
> (add-numbers (list 1 2 3 4))
10
```

We have even completely abstracted the dependency on numbers.

Power of functional abstraction

- General: Function that incrementally links elements of a list and a specified value

```
(define (op-elements op z l)
  (cond
  [(empty? l) z]
  [else
   (op (first l)
     (op-elements op z (rest l))))))
> (op-elements + 0 (list 5 8 12))
```

op: +
z: 0
l: (list 5 8 12)

(+ 5 (op-elements + 0 (list 8 12))

Power of functional abstraction

- General: Function that incrementally links elements of a list and a specified value

```
(define (op-elements op z l)
  (cond
  [(empty? l) z]
  [else
    (op (first l)
      (op-elements op z (rest l))))))
> (op-elements + 0 (list 5 8 12))
```

op: +
z: 0
l: (list 8 12)

(+ 8 (op-elements + 0 (list 12))

Power of functional abstraction

- General: Function that incrementally links elements of a list and a specified value

```
(define (op-elements op z l)
  (cond
  [(empty? l) z]
  [else
    (op (first l)
      (op-elements op z (rest l)))]))
> (op-elements + 0 (list 5 8 12))
```

op: +
z: 0
l: (list 12)

(+ 12 (op-elements + 0 (list)))

Power of functional abstraction

- General: Function that incrementally links elements of a list and a specified value

```
(define (op-elements op z l)
  (cond
  [(empty? l) z]
  [else
    (op (first l)
      (op-elements op z (rest l))))))
> (op-elements + 0 (list 5 8 12))
```

op: +
z: 0
l: (list)

0

Power of functional abstraction

- General: Function that incrementally links elements of a list and a specified value

```
(define (op-elements op z l)
  (cond
  [(empty? l) z]
  [else
   (op (first l)
     (op-elements op z (rest l)))]))
> (op-elements + 0 (list 5 8 12))
```

op: +
z: 0
l: (list 12)

(+ 12 0)

Power of functional abstraction

- General: Function that incrementally links elements of a list and a specified value

```
(define (op-elements op z l)
  (cond
  [(empty? l) z]
  [else
    (op (first l)
      (op-elements op z (rest l)))]))
> (op-elements + 0 (list 5 8 12))
```

op: +
z: 0
l: (list 8 12)

(+ 8 12)

Power of functional abstraction

- General: Function that incrementally links elements of a list and a specified value

```
(define (op-elements op z l)
  (cond
  [(empty? l) z]
  [else
    (op (first l)
      (op-elements op z (rest l)))]))
> (op-elements + 0 (list 5 8 12))
```

op: +
z: 0
l: (list 5 8 12)

(+ 5 20)

Power of functional abstraction

- General: Function that incrementally links elements of a list and a specified value

```
(define (op-elements op z l)
  (cond
  [(empty? l) z]
  [else
    (op (first l)
      (op-elements op z (rest l))))))
> (op-elements + 0 (list 5 8 12))
25
```

Power of functional abstraction

```
> (op-elements string-append "" (list "ab" "cd" "ef"))
"abcdef"
```

```
> (op-elements beside empty-image (list
     (circle 10 "solid" "red")
     (rectangle 10 10 "solid" "blue")
     (circle 10 "solid" "green")))
```

Copying a list per se is not interesting. Variants are: Creation of lists based on lists.

```
> (op-elements cons empty (list 5 8 12 2 9))
(list 5 8 12 2 9)
```

Power of functional abstraction

```
(define (append-list l1 l2)
  (op-elements cons l2 l1))
> (append-list (list 1 2) (list 3 4))
  (list 1 2 3 4)
```

Merging lists

Result on recursion termination.

Result with recursive call:
List consisting of the first element
and the result of the recursive call.

Power of functional abstraction

```
> (op-elements
    append-list
    empty
    (list (list 1 2) (list 3 4) (list 5 6)))
(list 1 2 3 4 5 6)
```

Flattening a list of lists.

Result on recursion termination.

Result with recursive call:
Merge the list in the first element and the list in the result of the recursive call.

Sorting with functional abstraction

```
; A (sorted-list-of Number) is a (list-of Number)
which is sorted by "<"

(list-of Number) -> (sorted-list-of Number)
; returns a list containing all elements of l sorted by "<"
(define (sort-list l) (op-elements insert empty l))

Number (sorted-list-of Number) -> (sorted-list-of Number)
; inserts x into a sorted list xs
(check-expect (insert 5 (list 1 4 9 12)) (list 1 4 5 9 12))
(define (insert x xs) ...)
```

Sorting with functional abstraction

```
Number (sorted-list-of Number) -> (sorted-list-of Number)
; inserts x into a sorted list xs
(check-expect (insert 5 (list 1 4 9 12)) (list 1 4 5 9 12))
(define (insert x xs)
  (cond [(empty? xs) (list x)]
        [(cons? xs) (if (< x (first xs))
                        (cons x xs)
                        (cons (first xs) (insert x (rest xs))))]))
```

Base case (recursion termination)

Recursive case

Recursive function call

Reuse through abstraction

- Avoids redundancy
- Promotes readability: complexity is concealed
- Saves work
- Reuse of properties such as correctness
  - Reused code has already been tested
  - More clients means more tests

Types of functions

- We have seen:
  Functions can process values of different types

- How can we indicate this in the signature?

```
; ...
; (list-of String) -> String
; (list-of Number) -> Number
(define (second l) (first (rest l)))
```

We could list all possible signatures.

Result: second element of the list regardless of the element type

Types of functions

- Enumerating signatures is not a good idea
  - Impossible to write them all down:
    - Data types can be defined as required
      → infinite number of signatures
  - Impractical to write down the ones used
    - Existing code must be adapted for new use

- Better option: type variables

Type variables

- Signature with type variables stands for the set of signatures resulting from all valid substitutions
  - Replacement by concrete type
  - The same type for all occurrences of the type variable

Declaration of the type variables

Use of the type variables

- Example

```
X] (list-of X) -> X
(define (second l) (first (rest l)))
```

Use of the type variables

Type variables

- All possible substitutions must result in a valid signature

- Is this signature with type variables correct?

```
; [X Y] (list-of X) -> Y
(define (second l) (first (rest l)))
```

Type variables

- All possible substitutions must result in a valid signature

- Is this signature with type variables correct?

```
; [X Y] (list-of X) -> Y
(define (second l) (first (rest l)))
```

No. A possible but invalid substitution is:
(list-of Number) -> String

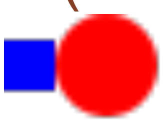
Type of functions

- Functions can be passed as an argument when calling a function
  - What is the type of the corresponding parameter?
  - How do we write the signature of functions with function parameters?

```
(define (add-circle f img)
  (f img (circle 10 "solid" "red")))
```

What is the signature of add-circle?

```
> (add-circle beside (rectangle 10 10 "solid" "blue"))
```

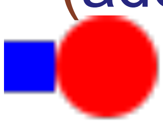


```
> (add-circle above (rectangle 10 10 "solid" "blue"))
```

Type of functions

- We already know a description for functions based on types: the signature
- We can consider the function signature as its type

```
(Image Image -> Image) Image -> Image
(define (add-circle f img)
  (f img (circle 10 "solid" "red")))
> (add-circle beside (rectangle 10
```



```
> (add-circle above (rectangle 10 10 "solid" "blue"))
```

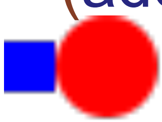
Signature from beside:
(Image Image -> Image)

Signature from above:
(Image Image -> Image)

Type of functions

- We already know a description for functions based on types: the signature
- We can consider the function signature as its type

```
(Image Image -> Image) Image -> Image
(define (add-circle
   (f img (circle 10 "
> (add-circle besid                      ue"))
```



The first argument must be a function with the signature (Image Image -> Image).

```
> (add-circle above (rectangle 10 10 "solid" "blue"))
```

Functions as return value

- We can pass functions as arguments
- A function can also return a function

```
Color -> Image
(define (big-circle color) (circle 20 "solid" color))

Color -> Image
(define (small-circle color) (circle 10 "solid" color))

String -> (Color -> Image)
(define (get-circle-maker size)
   (cond [(string=? size "big") big-circle]
         [(string=? size "small") small-circle]))
```

Function signature as the type of the return value

Function as return value

Functions as return value

- Calling a function with function as return value
  - Call comes first in a function call expression

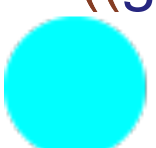
- Example:

```
String -> (Color -> Image)
(define (get-circle-maker size) ... )
```

expression results in a function

Calling the result function and passing an argument

```
> ((get-circle-maker "big") "cyan")
```

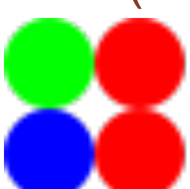


Note the brackets!

Nested function types

- For function type for parameter or return value
  - Parameter types and return types of this function can in turn be functions
- Nesting can be as deep as desired

```
((Image Image -> Image) Image -> Image) Image Image -> Image
(define (add-two-imgs-with f img1 img2)
  (above (f beside img1) (f beside img2)))
> (add-two-imgs-with
     add-circle
     (circle 10 "solid" "green")
     (circle 10 "solid" "blue"))
```



Use of the parameter f.
1st argument: Function
(Image Image -> Image)
2nd argument: Image
Return type: Image

Order of functions

- Order of functions indicates how strongly the signature is nested
  - "First-order function" ("first-order function")
    - Parameter and result types are not functions
    - Only a single arrow in signature
  - Second order function
    - At least one parameter or result type is the signature of a first-order function
  - And so on
- As a rule, a distinction is only made between
  - First-order functions (first-order functions)
  - Higher-order functions (Higher-Order Functions)

Order of functions

- Order of functions indicates how strongly the signature is nested
  - "First-order function" ("first-order function")
    - Parameter and result types are not functions
    - Only a sing
  - Second orde
    - At least on first-order function
  - And so the
- As a rule, a d
  - First-order functions (first-order functions)
  - Higher-order functions (Higher-Order Functions)

Key distinguishing feature for programming languages:

- Support for first-order functions only
- Support also for higher order functions

Polymorphic functions

- Functions whose result type depends on the arguments passed are called "polymorphic functions"
  - Example:

```
X] (list-of X) -> X
(define (second l) (first (rest l)))
```

- Higher-order functions can also be polymorphic
  - Example:
  - (define (op-elements op z l) ...)

What is the signature here?

Polymorphic functions

```
; [A B C D] A B C -> D
(define (op-elements op z l)
  (cond
    [(empty? l) z]
    [else
      (op (first l)
      (op-elements op z (rest l)))]))
```

Starting point: one type variable per parameter / return type

Now: Recognize dependencies.

Polymorphic functions

```
; [A B C D] A B C -> B
(define (op-elements op z l)
  (cond
   [(empty? l) z]
   [else
    (op (first l)
    (op-elements op z (rest l)))]))
```

z can be a return value. The return type must therefore correspond to the type of z.

Polymorphic functions

```
; [A B C D E B C - (list-of E)
(define (op-elements op z l)
  (cond
   [(empty? l) z]
   [else
    (op (first l)
    (op-elements op z (rest l))))))
```

l is passed to empty?, first, rest. It must therefore be a list.

Polymorphic functions

```
; [A B C D E] (E B -> B) -> B
(define (op-elements op z l)
  (cond
   [(empty? l) z]
   [else
    (op (first l)
    (op-elements op z (rest l))))))
```

A little more tidying up

op is called with the first element of l and the result of the recursive call. And its result can be the result of op-elements.

Polymorphic functions

```
X Y] (X Y -> Y) Y (list-of X) -> Y
(define (op-elements op z l)
  (cond
    [(empty? l) z]
    [else
      (op (first l)
      (op-elements op z (rest l))))))
```

Signature of polymorphic functions

- Be careful when deriving the signature from examples
  - (op-elements + 0 (list 5 8 12))
  - (op-elements string-append "" (list "ab" "cd" "ef"))

- In these examples, the signature appears to be:
  [X] (X X -> X) X (list-of X) -> X
  - This signature is valid, but too limited
  - However, another valid example is:
    - (op-elements cons empty (list 5 8 12 2 9))
    - Signature:

(Number (list-of Number) -> (list-of Number)) (list-of Number) (list-of Number) -> (list-of Number)

  - The previously determined signature is valid and as general as possible:
    [X Y] (X Y -> Y) Y (list-of X) -> Y

Type variables for data definitions

- For generic functions or data definitions
  - Several types are supported for certain elements
  - To list all the variants for this ...
  - ... Inserting a type variable
- Type variables for signatures
  - Declared by preceding square brackets
- Type variables for data definitions
  - Declared by use in the name of the data definition

```
; [X] a (list-of X) is either
; - empty
; - (cons X (list-of X))
```

Makes it clear that X is a type variable.

Type variables for data definitions

- Properties can be expressed using type variables

  ; [X] a (nonempty-list-of X) is: (cons X (list-of X))

Both the first element and the elements in the remainder list have the same type.

Grammar of types and signatures

- **<type>** ::= **<basic type>**
- **<data type**
- | '(' **<type constructor> <type>+** ')'
- | '(' **<type>+** '->' **<type>** ')'
- | **<X>**
- | '[' **<X>+** ']' **<type>**
- <Basic **type>** ::= 'Number'
- | String'
- | Boolean'
- | 'Image'
- <data **type>** ::= 'Posn'
- | 'WorldState'
- | ...
- **<TypeConstructor>** ::= 'list-of'
- | 'tree-of'
- | ...
- **<X>** ::= 'X'
- | 'Y'
- | ...

Types are constructed recursively.

Signatures are types

Type variables are types. But be careful: their use only makes sense if they are declared beforehand.

Philipps Universität Marburg

Function types

- We now know the syntax for declaring types, including function types
- Meaning of a type:
  - Set of all values with common properties
  - ... via which common functions are defined
- What is the meaning of a function type?
  - Informal: Set of all functions with corresponding signature

Philipps Universität Marburg