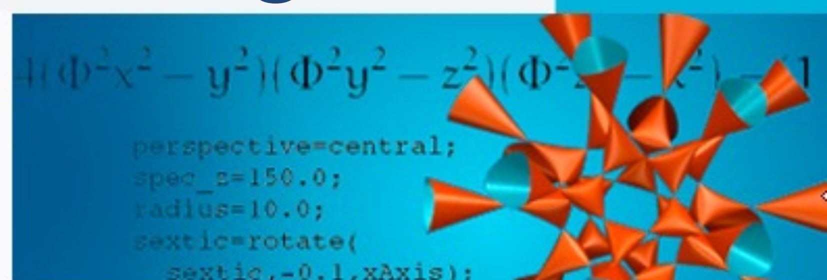
Philipps Universität Marburg

Declarative programming

Summer semester 2024

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

Imke Gürtler, Daniel Hinkelmann, Aaron Schafberg, Stefan Störmer



[Script 5.3 - 6]

Sum types

- So far:
  - Either primitive type
  - Or enumeration type
  - Or interval type
- Sum types
  - Combination of types
  - value of a sum type belongs to exactly one of the alternatives

Sum types

- Example: Function string->number
  - Result either number or false
  - Data type for result

```
A NorF is one of:
; - a Number
; - false

String -> NorF
; converts the given string into a number;
produces false if impossible
(define (string->number str) ...)
```

Sum types

- Association of types, therefore also
  - "Association type"
  - "Itemization"

- From the BSL documentation:

```
(string->number s) → (union number #false)        procedure
  s : string
```

Converts a string into a number, produce false if impossible.

```
> (string->number "-2.03")
-2.03
> (string->number "1-2i")
1-2i
```

Recipe: Design with sum types

- Example
  The tax on an item is either an absolute tax of 5 or 10 currency units, or a linear tax. Design a function that computes the price of an item after applying its tax.

- 1. information representation
  - Problem definition divides possible values into classes
  - Explicit specification of the classes

```
A Tax is one of
; - "absolute5"
; - "absolute10"
; - a Number representing a linear tax rate in percent
```

Recipe: Design with sum types

- 2. signature
  - Use of the sum type in signature

```
Number Tax -> Number
; computes price of an item after applying tax
(define (total-price itemprice tax) 0)
```

Recipe: Design with sum types

- 3. tests
  - At least one test per alternative
  - For alternative enumeration type: one test per value
  - For alternative interval type: Test for limits

```
(check-expect (total-price 10 "absolute5") 15)
(check-expect (total-price 10 "absolute10") 20)
(check-expect (total-price 10 25) 12.5)
```

Recipe: Design with sum types

- 4. splitting the main function (template)
  - Case differentiation of the alternatives
  - Condition can test for type
    Functions such as number? or string?

General: Structure of the function follows from the structure of the data.

```
(define (total-price itemprice tax)
   (cond [(number? tax) ...]
         [(string=? tax "absolute5") ...]
         [(string=? tax "absolute10") ...]))
```

Is the order of the cases significant?

Yes: only if the value is not a number may we perform a string comparison.

Recipe: Design with sum types

- 5. implement function body

- Implement cases of the template individually
- Focus on individual alternatives
- If necessary, simplification of the cond printout in post-processing

```
(define (total-price itemprice tax)
  (cond [(number? tax) (* itemprice (+ 1 (/ tax 100)))]
        [(string=? tax "absolute5") (+ itemprice 5)]
        [(string=? tax "absolute10") (+ itemprice 10)]))
```

Distinguishability of the cases

A Tax is one of

; - a Number representing an absolute tax in currency units

; - a Number representing a linear tax rate in percent

Can we treat this type of sum according to the given recipe?

No: Cannot decide for a value which alternative it belongs to.

Alternatives must be disjunctive. Otherwise, affiliation must be marked with a "tag".

Values with multiple properties

- Problem domain consists of entities
- Representation of entity by value
- Value has a data type

- So far:
  - Values are atomic
  - Represent exactly one property

Values with multiple properties

- In general, entities have several properties
- Nevertheless, representation by **a** value!

- Example:
  - Entity as the result of a function
  - WorldState with several properties

- Gödelization:
  - Coding of several properties as one atomic value
  - For example, as a character string
  - In practice: Conversion too time-consuming

Data type structures

- "Structures" or "Records"
  - Combination of several partial values into one value (date)
  - Each partial value can be accessed individually
- Example: Position structure
  - Partial values
    - x-coordinate
    - y-coordinate
  - Type of this structure: Number x Number

Borrowed from mathematics: cross product of quantities. Hence product type

- By the way:
  Sum types correspond to the union of sets: Tax: Number + String

Example

- BSL provides structure for the representation of positions

- A posn is a value
- posn has two components: x-coordinate, y-coordinate

- The structure is defined by functions

- Creation of instances: make-posn
- Signature: Number Number → Posn

```
(make-posn 3 4)
```

Design of structures

- Data types of the partial values
- Interpretation for the partial values

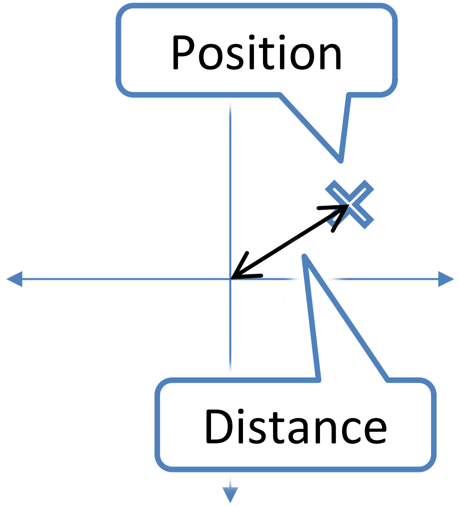
```
(define-struct posn (x y))
A Posn is a structure: (make-posn Number Number)
; interp. the number of pixels from left and from top
```

Instances of structures are values

- Use of instances of structures
  - As an argument
  - As return value
  - Just like other values (numbers, strings, etc.)

```
Posn -> Number
; to compute the distance of a-posn to the origin
(check-expect (distance-to-0 (make-posn 0 5)) 5)
(check-expect (distance-to-0 (make-posn 7 0)) 7)
(check-expect (distance-to-0 (make-posn 3 4)) 5)
(check-expect (distance-to-0 (make-posn 8 6)) 10)
(define (distance-to-0 a-posn) 0)
```

Just a stub!



Use of structure instances

- Structure is defined by certain functions:
  - Creation of instances
  - Access to components
  - Type test

- Access to x-coordinate: posn-x
- Signature Posn → Number

```
> (posn-x (make-posn 3 4))
3
```

- Access to y-coordinate: posn-y
- Signature Posn → Number

```
> (posn-y (make-posn 3 4))
4
```

Definition of structure types

- Not permanently installed (not primitive)
- User-defined
- Language provides a mechanism for definition

- BSL:

(define-struct *StructureName* (*FieldName*$_1$ ... *FieldName*$_2$))

- Example:

```
(define-struct posn (x y))
```

Definition of structure types

- When defining a structure, Racket automatically provides auxiliary functions

- Constructor
  - Creates instance of the structure
  - Name: *make-StructureName*
- Selectors for each partial value
  - Extracts the specified component from the structure instance
  - Name: *StructureName-FieldName*
- Structural predicate
  - Checks whether a value is an instance of the structure
  - Name: *StructureName?*

Nested structures

- Structures are data types
  - Can be used anywhere where data types are required
  - Also in the definition of structures

- Instances of structures are values
  - Can be used anywhere where values are required
  - Also in the definition of structure instances

Nested structures

- Example

```
(define-struct vel (deltax deltay))

; Posn Vel -> Posn
; computes position of loc after applying v
(check-expect (move (make-posn 5 6) (make-vel 1 2))
(make-posn 6 8))
(define (move loc v)
   (make-posn
      (+ (posn-x loc) (vel-deltax v))
      (+ (posn-y loc) (vel-deltay v))))
```

Expects position and movement. Actually two properties of an entity, e.g. ball.

Nested structures

- Example: Ball has the properties
  - x-coordinate, y-coordinate, delta-x and delta-y
  - Definition as a structure with four properties:

```
(define-struct ball (x y deltax deltay))
```

- Problem
  - Context of the fields is lost
  - Function move expected Posn and Vel
  - Instances of this must first be created

```
(move (make-posn (ball-x some-ball) (ball-y some-ball))
(make-vel (ball-deltax some-ball) (ball-deltay some-ball)))
```

Nested structures

- Better:
  - Use of the structures Posn and Vel in the definition of Ball
  - Preservation of logical structures
  - Reuse of auxiliary functions on substructures

```
(define-struct ball (loc vel))
(define some-ball
   (make-ball (make-posn 5 6) (make-vel 1 2)))
(move (ball-loc some-ball) (ball-vel some-ball))
```

Reuse of structures

```
(define-struct ball (loc vel))
; a Ball is a structure: (make-ball Posn Vel)
; interp. the position and velocity of a ball
```

Partial data:
Position and velocity specified in the comment.
Not binding.

- Assumption for definition of ball: 2-dimensional space
- Can we use other types instead of Posn and Vel can we use other types?
  - Number: Position in 1-dimensional space
  - Number: Velocity in 1-dimensional space

That works, but ...

```
; a Ball1d is a structure: (make-ball Number Number)
; interp. the position and velocity of a 1D ball
```

Reuse of structures

- Reuse possible with the same structure
- Joint development of
  - Position, Velocity, Ball
  - Pairs of values
  - E.g. possible to code all values as a position

```
; a Vel is a structure: (make-posn Number Number)
; interp. the velocity vector of a moving object

; a Ball is a structure: (make-posn Posn Vel)
; interp. the position and velocity of a ball
```

Reuse of structures

- For reuse
  - Data types cannot be differentiated by structure predicate
  - Not all functions via structure can be used:
    Expectations may be violated

```
; Ball -> Ball
; computes a new ball at a position of ball's posn after applying ball's vel
(define (move ball)
  (make-ball (make-posn (+ (posn-x (ball-loc ball)))
                           (vel-deltax (ball-vel ball)))
                        (+ (posn-y (ball-loc ball))
                           (vel-deltay (ball-vel ball))))
             (ball-vel ball)))
(define some-ball (make-ball (make-posn 5 6) (make-vel 1 2)))
>(move some-ball)
(make-ball (make-posn 6 8) (make-vel 1 2))
```

Reuse of structures

- For reuse
  - Data types cannot be differentiated by structure predicate
  - Not all functions via structure can be used: Expectations may be violated

```
; Ball -> Ball
; computes a new ball at a position of ball's posn after applying ball's vel
(define (move ball)
   (make-ball (make-posn (+ (posn-x (ball-loc ball)))
                            (vel-deltax (ball-vel ball)))
                         (+ (posn-y (ball-loc ball))
                            (vel-deltay (ball-vel ball))))
              (ball-vel ball)))
(define some-1d-ball (make-ball 1 2))
>(move some-1d-ball)
```

What happens if it is reused?

Reuse of structures

- For reuse
  - Data types cannot be differentiated by structure predicate
  - Not all functions via structure can be used:
    Expectations may be violated

```
; Ball -> Ball
; computes a new ball at a position of ball's posn after applying ball's vel
(define (move ball)
  (make-ball (make-posn (+ (posn-x (ball-loc ball)))
                           (vel-deltax (ball-vel ball)))
                        (+ (posn-y (ball-loc ball))
                           (vel-deltay (ball-vel ball))))
             (ball-vel ball)))
(define some-1d-ball (make-ball 1 2))
>(move some-1d-ball)
posn-x: expects a posn, given 1
```

What happens if it is reused?

Reuse of structures

- Reuse only with common semantic concept

- Or in the absence of a semantic concept
- Example Lisp: Representation of all product types by nested pairs

- (define-struct cons-cell (car cdr))

Any list of any values can be created using cons-cell. But: element type unknown.

First component: Value

Second component: Further cons-cell or "nil"

Philipps Universität Marburg

Design recipe with structures

- 1. information representation
- If the information description contains related data, this must be grouped as a structure.
- Field
  - Relevant property
  - Useful for all instances of the structure
  - Interpretation of the data
  - Description of which data is permitted
- Creating examples
  - For components with sum type:
    - Cover all values (enumeration type)
    - Covering all limits (interval type)

Design recipe with structures

- 3. tests
- For a function with product type as argument
- Use of example values from step 1
- For fields with sum type: amount of test data depending on enumerated values/intervals
- For product type: ideally test for every possible combination

Design recipe with structures

- Design recipe step 4: Template
  - Using the functions for extracting partial values

```
(define (distance-to-0 a-posn)
  (... (posn-x a-posn) ...
   ... (posn-y a-posn) ...))
```

- Implementation of the body function

```
(define (distance-to-0 a-posn)
  (sqrt
    (+ (sqr (posn-x a-posn))
       (sqr (posn-y a-posn)))))
```

Design recipe with structures

- 6. run tests
- Test directly after all headers have been written

- Tests must now fail (unless a dummy value happens to correspond to the expected value)

- This makes it possible to check that tests are not formulated too weakly

- The code must be completely covered during the tests (see code coloring in DrRacket)

Combination of totals and product types

```
(define-struct gcircle (center radius))
; A GCircle is (make-gcircle Posn Number)
; interp. the geometrical representation of a circle

(define-struct grectangle (corner-ul corner-dr))
A GRrectangle is (make-grectangle Posn Posn)
; interp. the geometrical representation of a rectangle
; where corner-ul is the upper left corner
; and corner-dr the down right corner

; A Shape is either:
; - a GCircle
; - a GRectangle
; interp. a geometrical shape representing a circle or a rectangle
```

Functions via totals and product types

- Main function overlaps/3: do three shapes overlap?

Template according to the design recipe.

```
; Shape Shape Shape -> Boolean
; determines whether the shapes overlap pairwise
(define (overlaps/3 shape1 shape2 shape3)
 (cond [(and (gcircle? shape1) (gcircle? shape2) (gcircle? shape3))
          ... overlaps-circle-circle-circle ...]
       [(and (gcircle? shape1) (gcircle? shape2) (grectangle? shape3))
          ... overlaps-circle-circle-rectangle ...]
       [(and (gcircle? shape1) (grectangle? shape2) (gcircle? shape3))
          ... overlaps-circle-rectangle-circle ...]
       ...)))
```

Functions via totals and product types

- Main function overlaps/3: do three shapes overlap?

```
; Shape Shape Shape -> Boolean
; determines whether the shapes overlap pairwise
(define (overlaps/3 shape1 shape2 shape3)
 (cond [(and (gcircle? shape1) (gcircle? shape2) (gcircle? shape3))
          ... overlaps-circle-circle-circle ...]
       [(and (gcircle? shape1) (gcircle? shape2) (grectangle? shape3))
          ... overlaps-circle-circle-rectangle ...]
       [(and (gcircle? shape1) (grectangle? shape2) (gcircle? shape3))
          ... overlaps-circle-rectangle-circle ...]
       ...)))
```

Template according to the design recipe.

Useful or necessary?

Neither ...

Algebraic data types

- Data types with a similar structure lead to similar auxiliary functions
- → DRY: don't repeat yourself!
- Abstraction through "algebraic data types"

Algebraic data types

- Circles and rectangles have common properties and operations
  - Enclosed area
  - Position can be changed
  - Resizable

- Superordinate concept: Form (or shape)

- Operations can generally be defined for higher-level types

```
; Shape Shape -> Boolean
; determines whether shape1 overlaps with shape2
(define (overlaps shape1 shape2) ...)
```

Algebraic data types

- Based on functions of the algebraic data type:
  - Development of further functions
  - Independent of specific type

```
; Shape Shape Shape -> Boolean
; determines whether the shapes overlap pairwise
(define (overlaps/3 shape1 shape2 shape3)
  (and
    (overlaps shape1 shape2)
    (overlaps shape1 shape3)
    (overlaps shape2 shape3)))
```

"Abstract algorithm"

Abstract algorithms

- Specific algorithms depend on specific data types
- Abstract algorithms are independent of a specific data type
- Very powerful code reuse
  - Abstract algorithms are immediately applicable to all variants (including new ones) of algebraic data types ...
  - ... as long as the corresponding concrete algorithms are implemented

Concrete algorithms on algebraic data type

- Checking all variants of the sum type

```
; Shape Posn -> Boolean
Determines whether a point is inside a shape
(define (point-inside shape point)
   (cond [(gcircle? shape) (point-inside-circle shape point)]
         [(grectangle? shape) (point-inside-rectangle shape point)]))
```

Why not write implementation directly here?

Concrete algorithms on algebraic data type

Why not write implementation directly in the cond expression?

- Variants of algebraic data type often product type
- Implementation of the individual cases then quickly becomes complex
- Outsourcing to auxiliary functions promotes reuse
- Two types of concrete algorithms
  1. For parameters of algebraic data type:
     Recognize the variant and forward ("Dispatch") to auxiliary function
  2. For parameters of concrete types (variants of the sum type):
     Implementation of the functionality for this type

Concrete algorithms on algebraic data type

- "Dispatch"

```
; Shape Posn -> Boolean
Determines whether a point is inside a shape
(define (point-inside shape point)
   (cond [(gcircle? shape) (point-inside-circle shape point)]
            [(grectangle? shape) (point-inside-rectangle shape point)]))
```

- Functionality

```
GCircle Posn -> Boolean
Determines whether a point is inside a circle
(define (point-inside-circle circle point)
   (<= (vector-length (posn- (gcircle-center circle) point))
      (gcircle-radius circle)))
```

Dispatch with multiple parameters of algebraic data type

```
; Shape Shape -> Boolean
; determines whether shape1 overlaps with shape2
(define (overlaps shape1 shape2)
   (cond [( and (gcircle? shape1) (gcircle? shape2))
             (overlaps-circle-circle shape1 shape2)]
            [( and (grectangle? shape1) (grectangle? shape2))
             (overlaps-rectangle-rectangle shape1 shape2)]
            [( and (grectangle? shape1) (gcircle? shape2))
             (overlaps-rectangle-circle shape1 shape2)]
            [( and (gcircle? shape1) (grectangle? shape2))
             (overlaps-rectangle-circle shape2 shape1 )]))
```

Dispatch with multiple parameters of algebraic data type

```
; Shape Shape -> Boolean
; determines whether shape1 overlaps with shape2
(define (overlaps shape1 shape2)
  (cond [( and (gcircle? shape1) (gcircle? shape2))
          (over
         [( and                                ? shape2))
          (over                                hape2)]
         [( and (grec          e? shape1) (gcircle? shape2))
          (overlaps-r ectangle-circle shape1 shape2)]
         [( and (gcircle? shape1) (grectangle? shape2))
          (overlaps-rectangle-circle shape2 shape1)]))
```

In the case of commutativity, for example, it is not necessary to implement a separate concrete function for each combination.

Specific functionality for specific data types

```
GCircle GCircle -> Boolean
; determines whether c1 overlaps with c2
(define (overlaps-circle-circle c1 c2)
   ; Two circles overlap if and only if the distance of their
   ; centers is smaller than the sum of their radii
   (<= (vector-length (posn- (gcircle-center c1)
                             (gcircle-center c2)))
       (+ (gcircle-radius c1) (gcircle-radius c2))))

GRectangle GRectangle -> Boolean
; determines whether r1 overlaps with r2
( define (overlaps-rectangle-rectangle r1 r2) ...)

GRectangle GCircle -> Boolean
; determines whether r overlaps with c
( define (overlaps-rectangle-circle r c) ...)
```

Design recipe with algebraic data types

- 1. information representation
- Various information is defined as (mostly) product type
- Represent common concept
- → Combine to algebraic data type using sum type

- Algebraic data types can be used wherever a type is required
  - Also as a variant of a totals type or in the field of a product type
  - → Hierarchical organization of algebraic data types

Hierarchical organization of algebraic data types

- Nested use of algebraic data types is good!
  - High degree of abstraction
  - More reuse

- Always try to combine common concepts into algebraic data types
  - → Grouping of alternatives of sum types:
    smaller number of alternatives
  - → Grouping of product type fields:
    smaller number of fields
  - Improved readability

Design recipe with algebraic data types

- 3. tests
- Algebraic data types are sum types at the highest level
- Therefore: at least one test per alternative
- Challenge: nested use of algebraic data types
  - → generally exponential increase in possible combinations with depth of hierarchy

Design recipe with algebraic data types

- 4. stencil
- Algebraic data type as parameter type, general: Sum type of product types
- Is it possible to formulate the function abstractly?
  - Implementation by calling existing functions
- Otherwise
  - Case differentiation of the alternatives with cond
  - Per case: Calling an auxiliary function for a specific type
  - For several parameters with algebraic data type: **Auxiliary function per combination**
  - At least for product types: Never implement directly!

Design recipe with algebraic data types

I.e. exclusively by calling functions that make sense on all values of the algebraic data type.

- 4. stencil
- Algebraic data type as parame
  Sum type of product types
- Is it possible to formulate the function abstractly?
  - Implementation by calling existing functions
- Otherwise
  - Case differentiation of the alternatives with cond
  - Per case: Calling an auxiliary function for a specific type
  - For several parameters with algebraic data type: **Auxiliary function per combination**
  - At least for product types: Never implement directly!

Design recipe with algebraic data types

- 7. post-processing
- Standardize commonalities in the variants
  - E.g. same representation of the same data
  - How to share auxiliary functions
- Avoid broad, flat algebraic data types
  - Grouping alternatives into sum types
  - Grouping fields into product types
- Logical grouping
  - Increases readability
  - Increases reusability