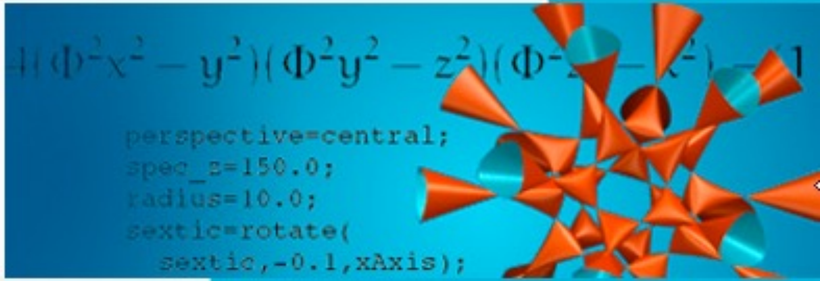
Philipps Universität Marburg

Declarative programming

Summer semester 2024

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

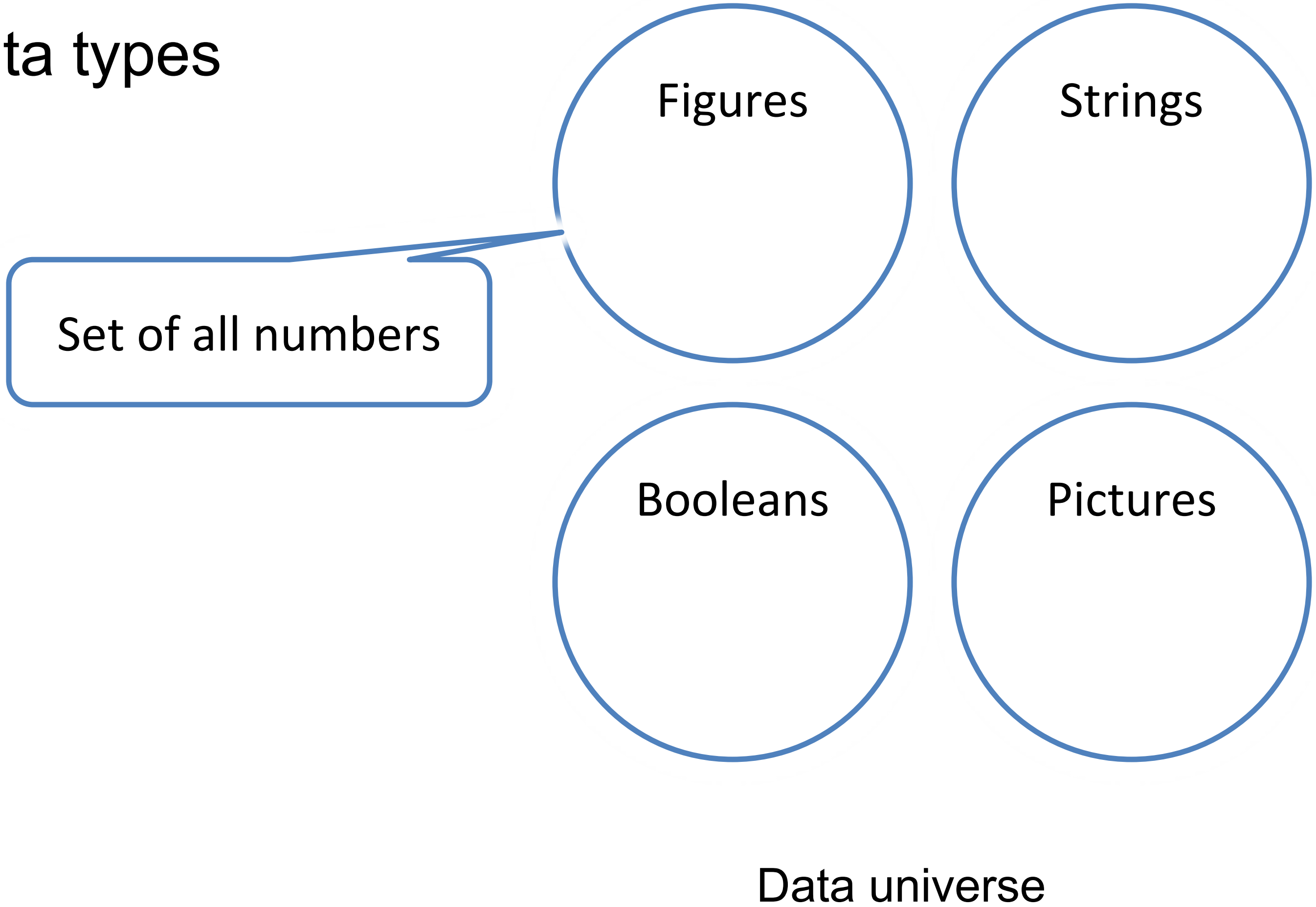
Imke Gürtler, Daniel Hinkelmann, Aaron Schafberg, Stefan Störmer



[Script 8.11, 9]

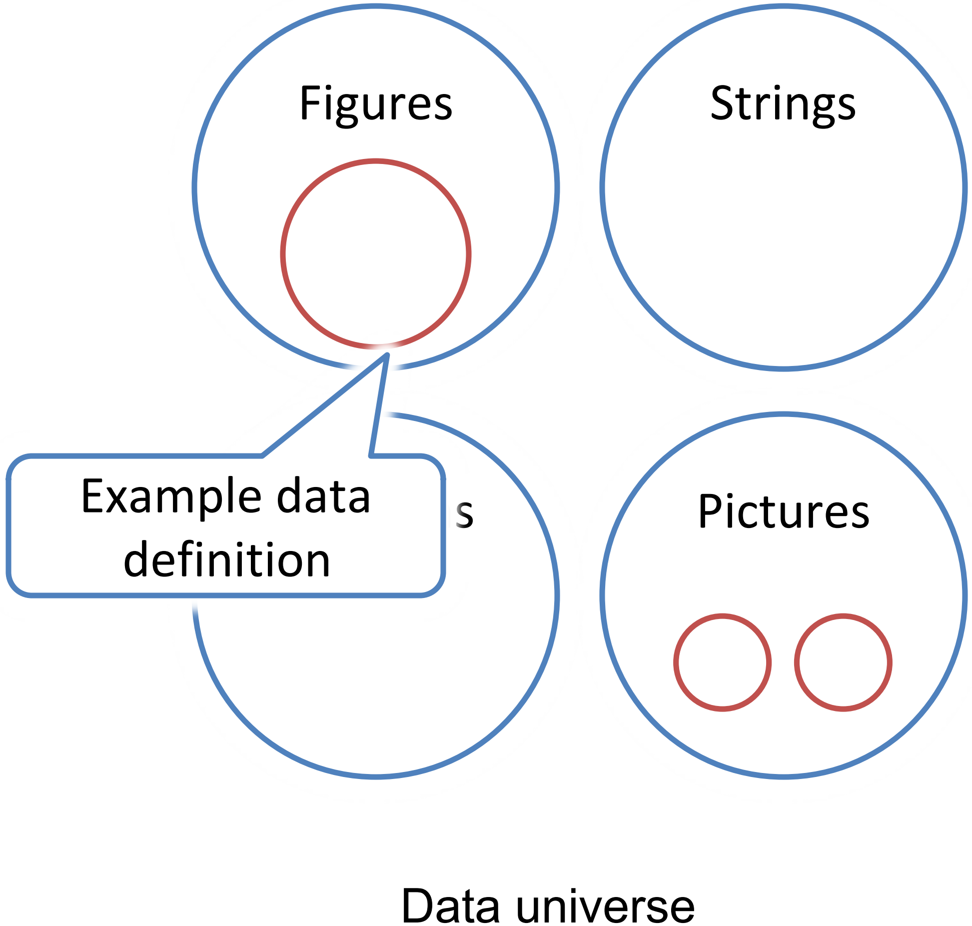
Data universe

- Primitive data types



Data universe

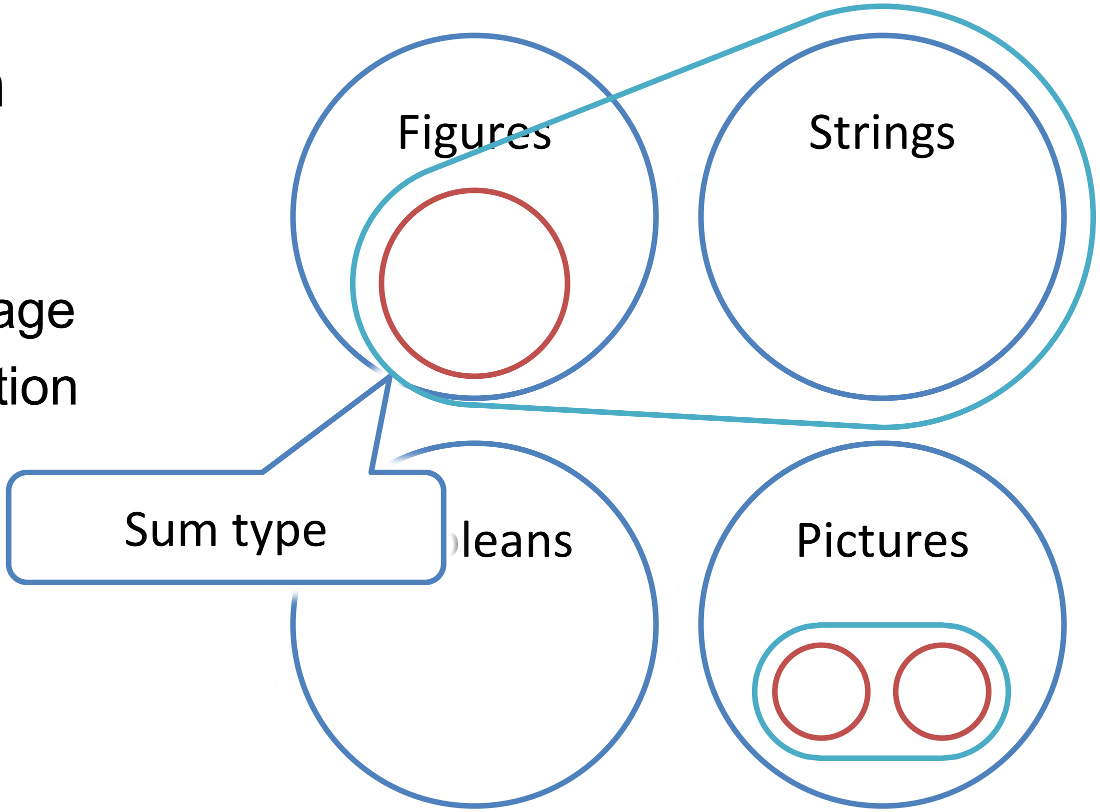
- Simple data definition
  - As comment
  - No precise meaning for programming language
  - Documented interpretation for programmers
- Restriction to subsets



Data universe

- Simple data definition
  - As comment
  - No precise meaning
    for programming language
  - Documented interpretation
    for programmers

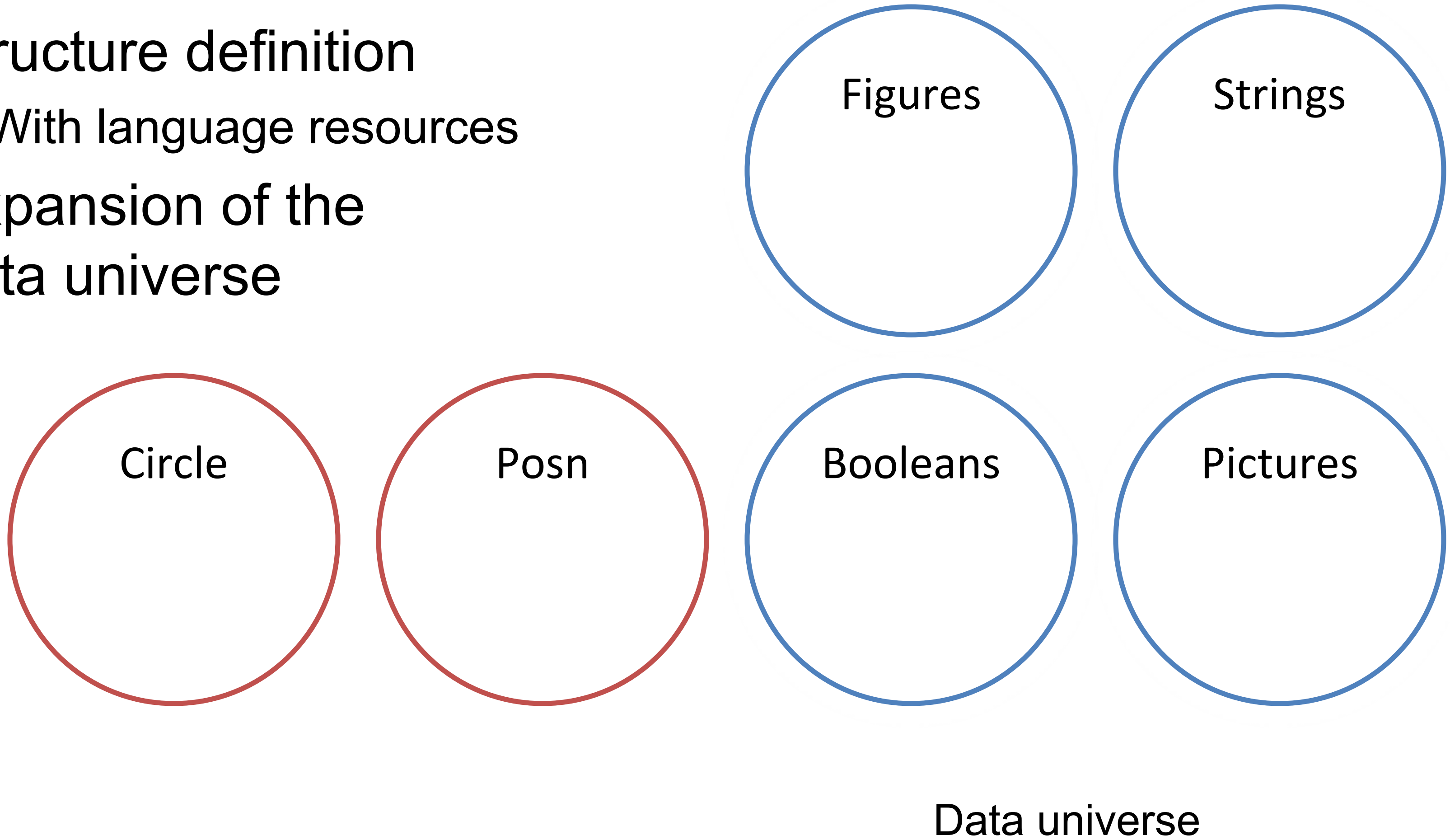
- Combined quantities



Data universe

Data universe

- Structure definition
  - With language resources
- Expansion of the data universe



Syntax vs. semantics

- Data universe
  - All values that can be formed by grammar <v>
- Forced restriction of language:
  - Only instances of defined structures
- Restriction through interpretation
  - Data definition specifies which values may be used for structure fields

Example

(define-struct posn (x y))

A Posn is a structure: (make-posn Number Number)

; interp. the number of pixels from left and from top

<make-undefined 3 4 >

Part of the data universe but not permitted by the language: Structure undefined not defined
No corresponding expression (make-undefined 3 4)

<make-posn true "x" >

Part of the data universe but Contradicts interpretation.
Therefore not part of the value set of the data type.

<make-posn 3 4 >

Part of the data universe and the value set (according to interpretation).

Data definition

- A data definition: coherent subset of the data universe

- Function signature
  - Which values from the data universe are accepted as arguments?
  - Which values from the data universe are produced as a result?

"Calculating" with types

- From the perspective of typisomorphy
  - Names do not matter
  - Only types play a role
  - General notation
    - Product types: (* String String Number)
    - Sum type: (+ ...)
    - Isomorphism: =
- Example total type

(+ Student Professor ResearchAssociate)

A UniversityPerson is either:

; - a student

; - a Professor

; - a ResearchAssociate

"Calculating" with types

- Known calculation rules apply

Associativity of *

(* X (* Y Z)) = (* (* X Y) Z) = (* X Y Z)

; Commutativity of *

(* X Y) = (* Y X)

; Associativity of +

(+ X (+ Y Z)) = (+ (+ X Y) Z) = (+ X Y Z)

; Commutativity of +

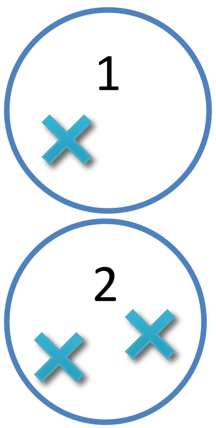
(+ X Y) = (+ Y X)

Distributivity of * and +

(* X (+ Y Z)) = (+ (* X Y) (* X Z))

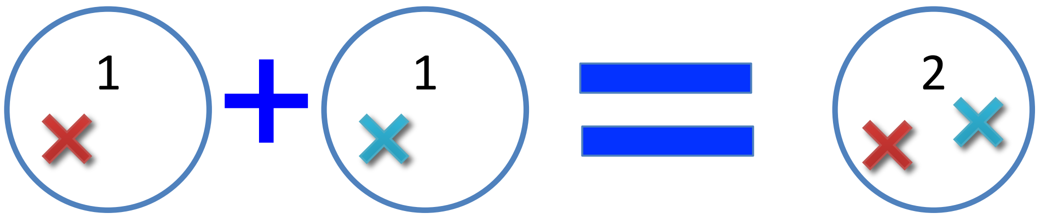
"Calculating" with types

- Sum types as tagged unions
  - Alternatives can always be distinguished
- Then analogy goes even further
- We call
  - A data type with a value of 1
  - A data type with two values 2

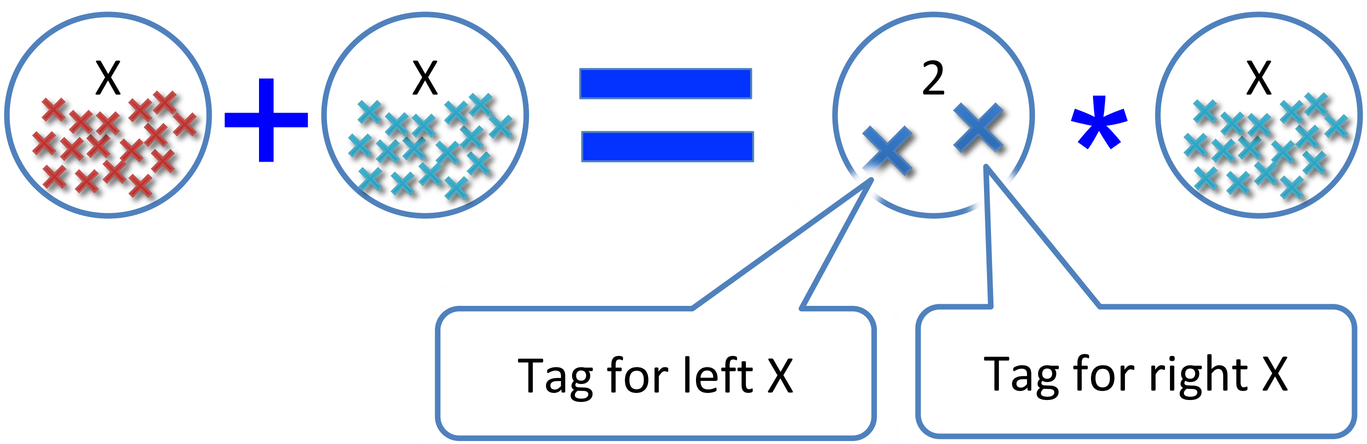


"Calculating" with types

- (+ 1 1) = 2



- (+ X X) = (* 2 X)



"Calculating" with types

- Cardinality of types
  - |X| is the cardinality of type X
  - Number of values of type X
- Calculating with cardinalities
  - |X + Y| = |X| + |Y|
  - |X * Y| = |X| * |Y|

Refactoring of data types

- (define-struct student1 (lastname firstname matnr))
- ; a Student1 is: (make-student1 String String Number)
- ; interp. lastname, firstname, and matrikel number of
- ; a student
- (define-struct student2 (matnr lastname firstname))
- ; a Student2 is: (make-student2 Number String String)
- ; interp. matrikel number, lastname, and firstname of
- ; a student
- (define-struct fullname (firstname lastname))
- ; a FullName is: (make-fullname String String)
- ; interp. first name and last name of a person
- (define-struct student3 (fullname matnr))
- ; a Student3 is: (make-student3 FullName Number)
- ; interp. full name and matrikel number of a student

Task: find an example of a student that can be represented in student1, -2 or -3, but not in the other structures.

Refactoring of data types

- (define-struct student1 (lastname firstname matnr))
- ; a Student1 is: (make-student1 String String Number)
- ; interp. lastname, firstname, and matrikel number of
- ; a student
- (define-struct student2 (matnr lastname firstname))
- ; a Student2 is: (make-student2 Number String String)
- ; interp. matrikel number, lastname, and firstname of
- ; a student
- (define-struct fullname (firstname lastname))
- ; a FullName is: (make-fullname String String)
- ; interp. first name and last name of a person
- (define-struct student3 (fullname matnr))
- ; a Student3 is: (make-student3 FullName Number)
- ; interp. full name and matrikel number of a student

Task: find an example of a student that can be represented in student1, -2 or -3, but not in the other structures.

There is **no** example!

Refactoring of data types

- In the example:
  - All representations can display the same information
  - Functions can be written that map an instance of one structure to an instance of another structure and vice versa

"Bijective mapping"

Student1 -> Student2

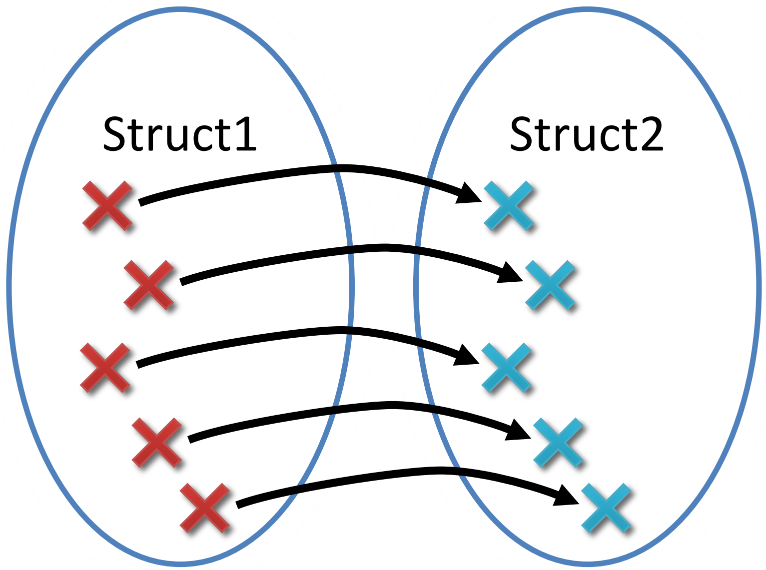
```
(define (Student1ToStudent2 s)
  (make-student2 (student1-matnr s) (student1-lastname s)
    (student1-firstname s)))
```

Student2 -> Student1

```
(define (Student2ToStudent1 s)
  (make-student1 (student2-lastname s) (student2-firstname s)
    (student2-matnr s)))
```

Isomorphism

- Bijective mapping:
  ; Struct1 -> Struct2
  (define (Struct1ToStruct2 s) ...)
  - Each value from Struct1 is assigned a value on Struct2 is assigned
  - Each value from Struct2 occurs as a result of occurs
- Two data types are isomorphic, if there is a bijective mapping between them



Refactoring with algebraic data types

- A data type can always be replaced by an isomorphic data type
  - Customize the constructor calls
  - Customizing the selector calls
- Isomorphism also when data is grouped

Student1 -> Student3

```
(define (Student1ToStudent3 s)
   (make-student3 (make-fullname (student1-firstname s)
      (student1-lastname s)) (student1-matnr s)))
```

Student3 -> Student1

```
(define (Student3ToStudent1 s)
   (make-student1 (fullname-lastname (student3-fullname s))
      (fullname-firstname (student3-fullname s))
         (student3-matnr s)))
```

Refactorings with algebraic data types

- Possible refactorings
  - Swapping sequences
  - "Inlining" of data types
  - "Outsourcing" of data types
  - "Multiplication" of products

Size of data

- So far:
  - Data is made up of a fixed number of atomic data

Posn consists of 2 numbers

1 Number

- Example

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

Circle is made up of 3 numbers

Data of any size

- The size of data often depends on the input

- Example: Family tree
  - Representation of a person's ancestor

```
(define-struct person (name father mother))
A Person is: (make-person String Parent Parent)
; interp. the name of a person with his/her parents

(define-struct parent (name grandfather grandmother))
A Parent is: (make-parent String GrandParend GrandParent)
; interp. the name of a person's parent
; with the names of his/her grandparents
```

A new structure must be defined for each generation.

Data of any size

```
(define-struct person (name father mother))
A Person is: (make-person String Parent Parent)
; interp. the name of a person with his/her parents

(define-struct parent (name grandfather grandmother))
A Parent is: (make-parent String GrandParend GrandParent)
; interp. the name of a person's parent
; with the names of his/her grandparents
```

Is there a problem with this definition?

Data of any size

```
(define-struct person (name father mother))
A Person is: (make-person String Parent Parent)
; interp. the name of a person with his/her parents

(define-struct parent (name grandfather grandmother))
A Parent is: (make-parent String GrandParend GrandParent)
; interp. the name of a person's parent
; with the names of his/her grandparents
```

Is there a problem with this definition?

Redundancy: Structure of person and parent is identical.

Self-similarity

- Data can be "self-similar"
  - One part has the same structure as the whole
  - Example: every person in every generation has a mother and a father
- Recursive data types
  - Possibility to define self-similar data

Recursive data types

```
(define-struct person (name father mother))
A FamilyTree is: (make-person String FamilyTree FamilyTree)
; interp. the name of a person and the tree of his/her parents.
```

Recursive data types

(define-struct person (name father mother))

A FamilyTree is: (make-person String FamilyTree FamilyTree)

; interp. the name of a person and the tree of his/her parents.

The definition for the data type refers to the data type itself.

- Previously: Previously defined data types may be used in the definition of a data type
- Now: Previously defined data types and the currently defined data type may be used in the definition of a data type

Recursive data types

- How do you create a value of a recursive data type?

```
(make-person "Heinz"
   (make-person "Horst"
      (make-person "Joe" ...)
      ...)
   ...)
```

How can this chain stop?

Recursive data types

- Recursion must be able to terminate
- Definition of FamilyTree as a sum type

```
(define-struct person (name father mother))
A FamilyTree is either:
; - (make-person String FamilyTree FamilyTree)
; - false
; interp. either the name of a person and the tree of its parents,
; or false if the person is not known/relevant.
```

Recursive data types

```
(define HEINZ
  (make-person "Heinz"
    (make-person "Elke" false false)
    (make-person "Horst"
                 (make-person "Joe"
                              false
                              (make-person "Rita"
                                           false
                                           false))
                 false)))
```

Values of a recursive data type

- Definition of the possible values "inductive"
- First step: Values that are not recursive:
  - ft_0 = { false }
- Subsequent steps: Values that can be constructed from values of the previous step
  - $ft_n = ft_{n-1} \cup$ { (make-person name p_1 p_2) | name ∈ Strings and p_1 ,p_2 ∈ ft $\}_{n-1}$

For example:
$ft_1 = ft_0 \cup$ { (make-person name false false) | name ∈ Strings }

- Set of values: ft_n for n → ∞
  - The depth of the recursion and therefore the number of values is unlimited

Functions via recursive data types

- Question: Does a person p have an ancestor with the name a?

- We say that a person is also an ancestor of themselves

- Procedure:
  - Does the person himself have the name?
  - Does the father or mother have the name?
  - And so on for their fathers and mothers

Functions via recursive data types

```
FamilyTree String -> Boolean
; determines whether person p has an ancestor a
(check-expect (person-has-ancestor HEINZ "Joe") true)
(check-expect (person-has-ancestor HEINZ "Emil") false)
(define (person-has-ancestor p a)
  (cond [(person? p)
          ...(person-name p) ...
          ...(person-father p)...
          ...(person-mother p) ...]
        [else ...]))
```

Design recipe for sum type

Design recipe for product type

Functions via recursive data types

- According to the design recipe:
  - Help functions for all fields that have a complex type themselves

```
FamilyTree String -> Boolean
; determines whether person p has an ancestor a
(define (person-has-ancestor p a)
   (cond [(person? p)
            ... (person-name p) ...
            ... (father-has-ancestor (person-father p) ...)...
            ... (mother-has-ancestor (person-mother p) ...)...]
          [else ...]))
```

What should father-has-ancestor and mother-has-ancestor look like?

Functions via recursive data types

```
FamilyTree String -> Boolean
; determines whether father p has an ancestor a
(define (father-has-ancestor p a)
   (cond [(person? p)
             ... (person-name p) ...
             ... (grand-father-has-ancestor (person-father p) ...)...
             ... (grand-mother-has-ancestor (person-mother p) ...)...]
            [else ...]))
```

We have to define two auxiliary functions again.

It is noticeable that the signature and meaning of the functions are identical.

Functions via recursive data types

- When processing self-similar partial data, the function itself can be called

```
(define (person-has-ancestor p a)
  (cond [(person? p)
            ... (person-name p) ...
            ... (person-has-ancestor (person-father p) ...)...
            ... (person-has-ancestor (person-mother p) ...)...]
           [else ...]))
```

Functions via recursive data types

- As before: The structure of the data determines the structure of the function
  - For recursive data
  - If the functions are also recursive

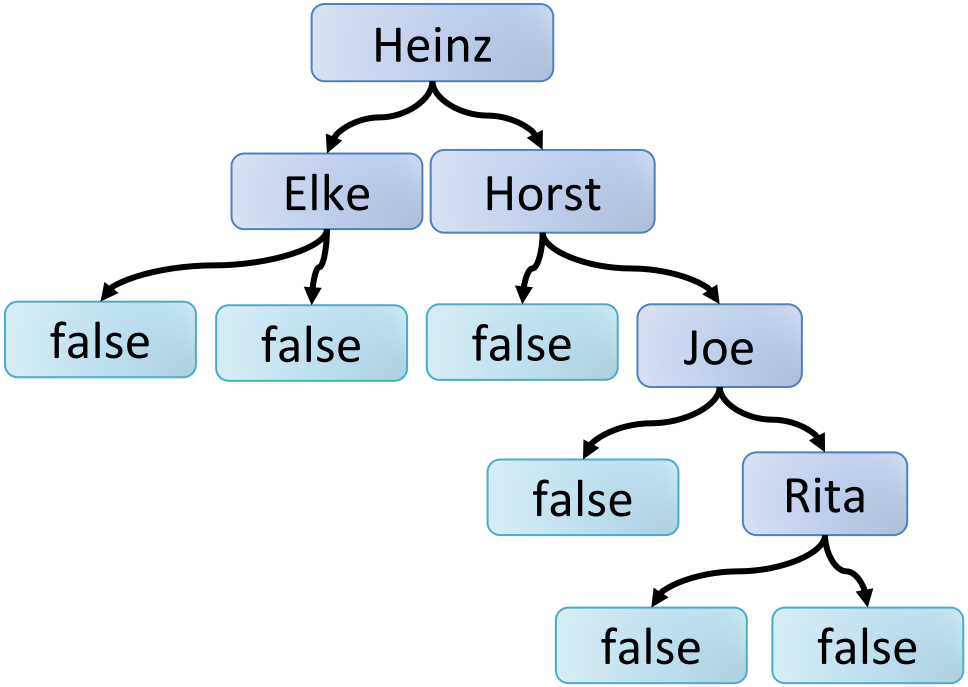
Functions via recursive data types

```
(define (person-has-ancestor p a)
  (cond [(person? p)
         (or
           (string=? (person-name p) a)
           (person-has-ancestor (person-father p) a)
           (person-has-ancestor (person-mother p) a))]
        [else false]))
```

Recursive data types

```
<make-person "Heinz"
  <make-person "Elke" false false>
  <make-person "Horst"
           <make-person "Joe"
                        false
                        <make-person "Rita"
                                     false
                                     false>>
           false>>>
```

Recursive data types



(person-has-ancestor HEINZ "Joe")

Functions via recursive data types

- Important: recursive calls may only be made for recursive partial data
- With each recursive call, the "problem" must become smaller, otherwise the recursion will not terminate
- Every recursive function must have a termination condition, i.e. a case without a recursive call

Philipps Universität Marburg

Functions via recursive data types

```
(define (person-has-ancestor p a)
  (cond [(person? p)
          (or
            (string=? (person-name p) a)
            (person-has-ancestor (person-father p) a)
            (person-has-ancestor (person-mother p) a))]
        [else false]))
```

Recursive call with partial data.

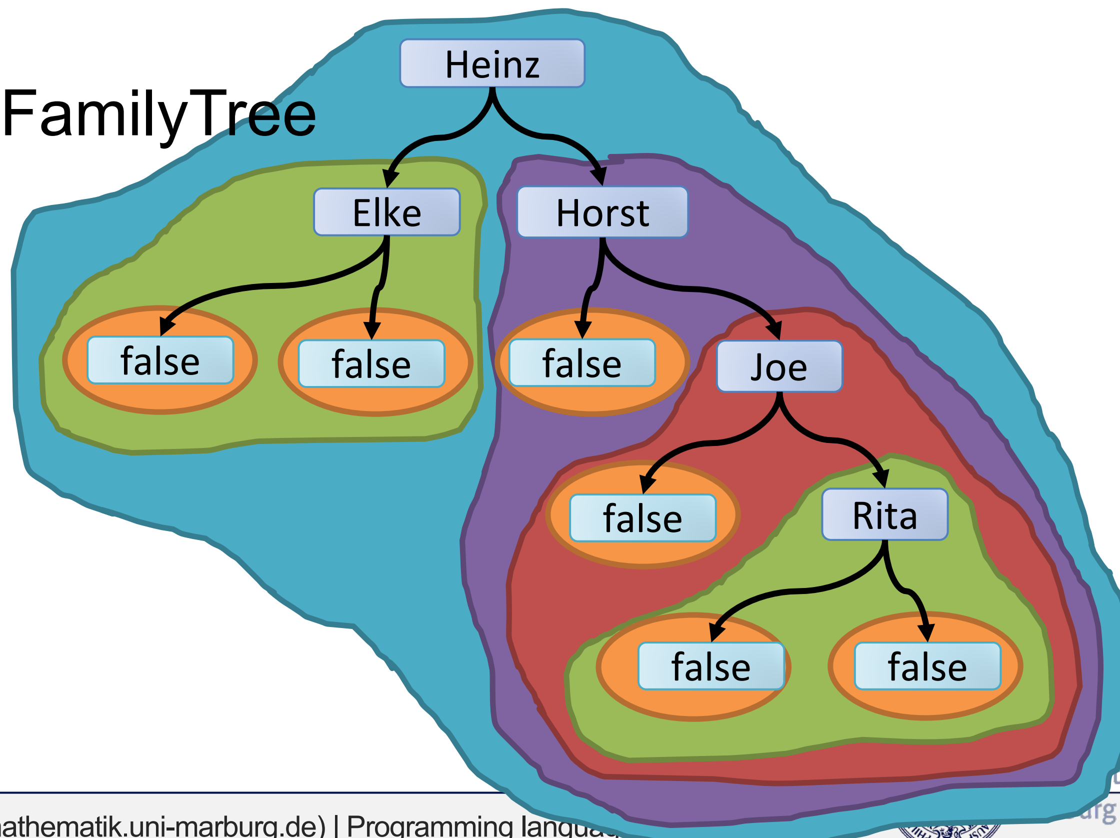
Termination condition

Termination of recursive functions

- "Induction proof"
  - Step-by-step proof of a property
  - Analogous to the step-by-step structure of the values of a recursive data type
- Using the example of FamilyTree
  - For each person p there is an i, such that p $\in ft_i$ and p $\notin ft_{i-1}$
  - The ancestors of p are then $\in ft_{i-1}$

$\in ft_0$ $\in ft_2$ $\in ft_4$

$\in ft_1$ $\in ft_3$



Termination of recursive functions

- "Induction proof"
  - Step-by-step proof of a property
  - Analogous to the step-by-step structure of the values of a recursive data type
- Using the example of FamilyTree
  - For each person p there is an i such that $p \in ft_i$ and $p \notin ft_{i-1}$
  - The ancestors of p are then $\in ft_{i-1}$
  - This means that the argument of the recursive calls is $\in ft_{i-1}$
  - For $p \in ft_0$ the function obviously terminates: ft_0 = { false }
  - This means that the function also terminates for $p \in ft_1$, $p \in ft_2$, etc.
- We have shown that the person-has-ancestor function is "well-defined"
  - It returns a result for every possible argument value

Termination of recursive functions

- Given another recursive function

```
FamilyTree -> Boolean
; determines whether person p has an ancestor a
(check-expect (person-has-ancestor-stupid HEINZ "Joe") true)
(check-expect (person-has-ancestor-stupid HEINZ "Emil") false)
(define (person-has-ancestor-stupid p a)
  (person-has-ancestor-stupid p a))
```

- Does this also schedule?
  - For $p \in ft_i$, the recursive call $p \in ft_i$ is also possible.
  - If $p \in ft_0$, i.e. p = false, then the reduction (and therefore the function) does not terminate
  - We have therefore shown that the function does not terminate

Structural recursion

- Recursive data
  - Indeterminate size
  - Finite size

- Therefore
  - If a recursive function follows the recursive structure of the data
  - And are the data well-defined
  - Then the function is well-defined

Producing instances of recursive data types

- For example, creating a modified family tree (adding a title)
- Procedure
  - Analogous to the previous function
  - Nested recursion
  - Adjusted termination case

```
(define (promote p t)
  (cond [(person? p)
         (make-person
           (string-append t (person-name p))
           (promote (person-father p) t)
           (promote (person-mother p) t))]
        [else p]))
```

Recursive call as argument of make-person.

Returns the current argument.