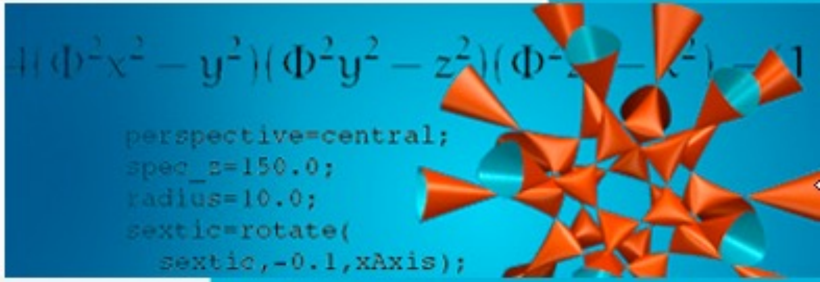
Philipps Universität Marburg

Declarative programming

Summer semester 2024

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

Imke Gürtler, Daniel Hinkelmann, Aaron Schafberg, Stefan Störmer



[Script 16,17]

Power of programming languages

- Any problem can be solved in any (general-purpose) programming language
- Differentiation of languages in support for design techniques
- Case study: Support for data definitions and signatures

Power of programming languages

- Comparison criteria
  1. Ensure that the use of values/functions matches their representation
  2. Time at which errors are found
  3. Reference from error message to error cause
  4. Expressiveness of the signature language
  5. Restriction of possible programs
  6. Consistency between data definitions/functions and program behavior

Philipps Universität Marburg

Classification of programming languages

- Type
  - Values have type
  - Functions expect types of arguments

- Classification according to handling types
  - Untypical languages
  - Dynamically typed languages
  - Dynamically verified signatures and contracts
  - Statically typed languages

Untypical languages

- Internally, all values are represented by numbers or blocks of numbers
- Therefore, for example, the addition of number and string is possible, even if it does not make sense
- Example: Assembler languages

Untypical languages

- Rating
  1. Ensure appropriate use of values/functions
     - Not through language. Only through coding discipline
  2. Time at which errors are found
     - Very late, i.e. when using incorrectly calculated values
  3. Reference from error message to error cause
     - Error typically occurs in program parts that are far away from the faulty calculation
  4. Expressiveness of the signature language
     - Not available
  5. Restriction of possible programs
     - No restrictions
  6. Consistency between data definitions/functions and program behavior
     - None

Dynamically typed languages

- Values have a type
- Assignment of value to type exists at runtime, can be queried

- Types of types
  - Built-in (primitive) types
  - User-defined types

- Example: Racket / Student Language
  - Query of the type e.g.: boolean?, number?, string?, symbol? or structure predicates

Dynamically typed languages

- For primitive operations
  - Runtime system checks applicability to values of the given types
- Example: (+ x y)
  - Check that x and y are numbers
  - In case of violation error message instead of e.g. interpreting Boolean value as number
- Does not (automatically) apply to user-defined data types and functions
  - We have so far only specified signatures as comments

Dynamically typed languages

- Type information only available for built-in types
  - For example, primitive types or structs
- No type information for only logically defined types
  - E.g. no differentiation whether number is used as temperature or length

Dynamically typed languages

```
Number (list-of Number) -> (list-of Number)
; returns the remainder of xs after first occurence of x, or empty otherwise
(define (rest-after x xs)
   (if (empty? xs)
      empty
      (if (= x (first xs))
         (rest xs)
         (rest-after x (rest xs)))))
> (rest-after 5 (list 1 2 3 4))
'()
> (rest-after 2 (list 1 2 3 4))
'(3 4)
```

Dynamically typed languages

```
Number (list-of Number) -> (list-of Number)
; returns the remainder of xs after first occurence of x, or empty otherwise
(define (rest-after x xs)
  (if (empty? xs)
    empty
    (if (= x (first xs))
      (rest xs)
      (rest-after x (rest xs)))))
> (rest-after 5 (list 1 2 3 4))
'()
> (rest-after 2 (list 1 2 3 4))
'(3 4)
> (rest-after 2 (list "one" "two" "three"))
=: expects a number as 2nd argument, given "one"
> (rest-after 2 (list 1 2 "three" "four"))
'("three" "four")
```

Definition of the signature

Signature violation detected during evaluation. Runtime error

Not all signature violations are detected during function execution.

Dynamically typed languages

```
Number (list-of Number) -> (list-of Number)
; returns the remainder of xs after first occurence of x, or empty otherwise
(define (rest-after x xs)
  (if (empty? xs)
    empty
    (if (= x (first xs))
      (rest xs)
      (rest-after x (rest xs)))))
> (rest-after 5 (list 1 2 3 4))
'()
> (rest-after 2 (list 1 2 3 4))
'(3 4)
> (rest-after 2 (list "one" "tw
=: expects a number as 2n
> (rest-after 2 (list 1 2 "thre
'("three" "four")
```

Violation of the signature by the caller

Dynamically typed languages

```
Number (list-of Number) -> (list-of Number)
; returns the remainder of xs after first occurence of x, or empty otherwise
(define (rest-after x xs)
  (if (empty? xs)
    "not a list"
    (if (= x (first xs))
      (rest xs)
      (rest-after x (rest xs)))))
> (rest-after 5 (list 1 2 3 4))
"not a list"
> (cons 6 (rest-after 5 (list 1 2 3 4)))
cons: second argument must be a list, but received 6 and "not a list"
```

Runtime error when using the result value

Error can occur much later. Therefore: Error cannot be traced back to rest-after

Philipps Universität Marburg

Dynamically typed languages

```
Number (list-of Number) -> (list-of Number)
; returns the remainder of xs after first occurence of x, or empty otherwise
(define (rest-after x xs)
  (if (empty? xs)
    "not a list"
    (if (= x (first xs))
      (rest xs)
      (rest-after x (rest xs)))))
> (rest-after 5 (list 1 2 3 4))
"not a list"
> (cons 6 (rest-after 5 (list 1 2 3 4)))
cons: second argument must be a list, but received 6 and "not a list"
```

Violation of the signature by the implementation

Dynamically typed languages

- Rating
  1. Ensure appropriate use of values/functions
     - Only for primitive (and struct) types and functions
  2. Time at which errors are found
     - Runtime
  3. Reference from error message to error cause
     - Error may occur in program parts that are far away from the incorrect calculation
  4. Expressiveness of the signature language
     - Not available
  5. Restriction of possible programs
     - As good as none
  6. Consistency between data definitions/functions and program behavior
     - Restricted

Dynamically verified signatures and contracts

- Definition of signatures and data definition as a program
- Verification of the signature at runtime

```
; a list-of-numbers is either:
; - empty
; - (cons Number list-of numbers)

X] (list-of X) -> Boolean
; checks whether xs contains only numbers
(define (list-of-numbers? xs)
  (if (empty? xs)
      true
      (and (number? (first xs))
           (list-of-numbers? (rest xs)))))
```

User-defined predicates

Dynamically verified signatures and contracts

- Checking the correct argument types
  - Use of primitive predicates together with user-defined predicates

```
Number (list-of Number) -> (list-of Number)
; dynamically checked version of rest-after
(define (rest-after/checked x xs)
  (if (number? x)
    (if (and (list? xs)
          (list-of-numbers? xs))
      (if (list-of-numbers? (rest-after x xs))
          (rest-after x xs)
          (error "function must return list-of-numbers"))
      (error "second arg must be list-of-numbers"))
  (error "first arg must be a number")))
```

Checking the argument and return types

If successful, call the function

Error messages

Dynamically verified signatures and contracts

```
> (rest-after/checked 2 (list 1 2 3 4))
'(3 4)
```

If used correctly: Behavior as before

```
> (rest-after/checked "x" (list 1 2 3 4))
first arg must be a number
```

If used incorrectly: Error message when calling the function

```
> (rest-after/checked 2 (list 1 2 "three" 4))
second arg must be list-of-numbers
```

Some previously successful function calls are now prevented

Dynamically verified signatures and contracts

```
> (rest-after/checked 2 (list 1 2 "three" 4))
second arg must be list-of-numbers
```

Some previously successful function calls are now prevented

Bad or good?

Prevents subsequent errors when using the result value

Dynamically verified signatures and contracts

- Laborious review of contracts
- Language for defining contracts
  - Instead of implementation in functional bodies
  - Specification of contracts at the interface

Condition for 1st argument

Condition for 2nd argument

```
(provide
  (contract-out
    [rest-after (-> number? (listof number?) (listof number?))]))
(define (rest-after x xs)
  (if (empty? xs)
      "not a list"
      (if (= x (first xs))
          (rest xs)
          (rest-after x (rest xs)))))
```

Condition for result value

Dynamically verified signatures and contracts

- Racket modular concept
  - File as module granularity
  - Module name: File name

Language setting!

heinz.rkt

```
#lang racket
(provide
   (contract-out
      [rest-after (-> number? (listof number?) (listof number?))]))
(define (rest-after x xs)
   (if (empty? xs)
       "not a list"
       (if (= x (first xs))
          (rest xs)
          (rest-after x (rest xs)))))
```

Dynamically verified signatures and contracts

- Racket modular concept
  - To use a module, it must be "imported" (require)

elke.rkt

```
#lang racket
(require "heinz.rkt")
(rest-after "x" (list 1 2 3 4))
```

Breach of contract

Dynamically verified signatures and contracts

```
(rest-after "x" (list 1 2 3 4))
```

```
rest-after: contract violation
   expected: number?
   given: "x"
   in: the 1st argument of
      (->
         number?
         (listof number?)
         (listof number?))
   contract from: /Users/klaus/heinz.rkt
   blaming: /Users/klaus/elke.rkt
      (assuming the contract is correct)
   at: /Users/klaus/heinz.rkt:3.24
```

Detailed explanation of the breach of contract

Responsible for breach of contract

Dynamically verified signatures and contracts

```
(rest-after 5 (list 1 2 3 4))

rest-after: broke its contract
  promised: "list?"
  produced: "not a list"
  in: the range of
    (->
      number?
      (listof number?)
      (listof number?))
  contract from: /Users/klaus/heinz.rkt
  blaming: /Users/klaus/heinz.rkt
    (assuming the contract is correct)
  at: /Users/klaus/heinz.rkt:3.24
```

Violation of the contract for return value is also checked

Responsible for breach of contract

Philipps Universität Marburg

Dynamically verified signatures and contracts

- Contracts are only checked when the program is executed
- No certainty that all contract violations will be found
  - New program execution may result in new contract violations
- Contracts can only check conditions that can be calculated

Dynamically verified signatures and contracts

- Rating
  1. Ensure appropriate use of values/functions
     - Yes
  2. Time at which errors are found
     - Runtime
  3. Reference from error message to error cause
     - Strong cover
  4. Expressiveness of the signature language
     - Limited to predictable conditions
  5. Restriction of possible programs
     - Stronger than dynamically typed signatures
  6. Consistency between data definitions/functions and program behavior
     - Provided the contract is correctly specified

Statically typed languages

- Type system
  - Verification of signatures and function calls before runtime
  - Review of all possible designs
- Compositionality
  - Check only depending on the module itself and the types/signatures of the directly used modules
- Properties
  1. If the type check is successful ("well-typed" program): A type error cannot occur during execution
  2. There are programs that are rejected even though there may be versions without type errors
     - Rice's theorem: non-trivial behavioral properties are not decidable

Statically typed languages

- Statically typed variant of Racket
  - Language level: Typed Racket

#lang typed/racket

```
(: rest-after (-> Integer (Listof Integer) (Listof Integer)))
(define (rest-after x xs)
  (if (empty? xs)
      empty
      (if (= x (first xs))
          (rest xs)
          (rest-after x (rest xs)))))
```

Static declaration of the function signature

Statically typed languages

```
> (rest-after 2 (list 1 2 3 4))
- : (Listof Integer)
'(3 4)

> (rest-after "x" (list 1 2 3 4))
eval:5:0: Type Checker: type mismatch
expected: Integer
given: String
in: "x"

> (rest-after 2 (list 1 2 "three" 4))
eval:6:0: Type Checker: type mismatch
expected: (Listof Integer)
given: (List One Positive-Byte String Positive-Byte)
in: (list 1 2 "three" 4)
```

Statically typed languages

- Type check also possible if program contains runtime errors

```
> (:print-type (rest-after (/ 1 0) (list 1 2 3 4)))
(Listof Integer)
```

Statically typed languages

- Type check of the function definition without call

```
(: rest-after (-> Integer (Listof Integer) (Listof Integer)))
(define (rest-after x xs)
   (if (empty? xs)
       "not a list"
       (if (= x (first xs))
           (rest xs)
           (rest-after x (rest xs)))))
eval:4:9: Type Checker: type mismatch
expected: (Listof Integer)
given: String
in: "not a list"
```

Statically typed languages

- Static type testing also possible if not all applications are known

- Reduction semantics for statically typed languages
  - Reduction preserves well-formedness ("Preservation" or "Subject Reduction" theorem)
  - Well-typed programs
    - Are either values or
    - Can always be reduced

Statically typed languages

- There are always programs that are rejected, although there may be versions without type errors

- Example
  - Dynamically typed: Execution without type error

```
> (+ 1 ( if (> 5 2) 1 "a"))
2
```

  - Statically typed: Type error

```
> (+ 1 ( if (> 5 2) 1 "a"))
eval:2:5: Type Checker: type mismatch
expected: Number
given: (U String One)
in: (if (> 5 2) 1 "a")
```

U: "Union type" or sum type

Statically typed languages

- Rating
  1. Ensure appropriate use of values/functions
     - Yes
  2. Time at which errors are found
     - Development time
  3. Reference from error message to error cause
     - Strong cover
  4. Expressiveness of the signature language
     - Formal syntax for signatures
  5. Restriction of possible programs
     - Greater restriction than with dynamically typed languages
  6. Consistency between data definitions/functions and program behavior
     - Runtime type errors can be excluded

Statically typed languages

- Rating
  1. Ensure appropriate use of values/functions
     - Yes
  2. Time at which errors are found
     - Development time
  3. Reference from error message to error cause
     - Strong cover
  4. Expressiveness of the signature language
     - Formal syntax for signatures
  5. Restriction of possible programs
     - Greater restriction than with dynamically typed languages
  6. Consistency between data definitions/functions and program behavior
     - Runtime type errors can be excluded

Trade-off between type safety and flexibility

Language support for algebraic data types

- Different ways of defining algebraic data types
  - Differ in properties, similar to support for signatures
- Algebraic data types have
  - Data definition
  - Interface: Set of constructors, selectors and predicates

Algebraic data type - example

```
An Expression is one of:
; - (make-literal Number)
; - (make-addition Expression Expression)
; interp. abstract syntax of arithmetic expr.

Number -> Expression
; constructs a literal exression
(define (make-literal value) ...)
Expression -> Number
; returns the number of a literal
; throws an error if lit is not a literal
(define (literal-value lit) ...)
X] X -> Bool
; returns true iff x is a literal
(define (literal? x) ...)
Expression Expression -> Expression
; constructs an addition expression
(define (make-addition lhs rhs) ...)
X] X -> Bool
; returns true iff x is an addition expr.
(define (addition? x) ...)
Expression -> Expression
; returns left hand side of an addition expression
; throws an error if e is not an addition expression
(define (addition-lhs e) ...)
Expression -> Expression
; returns right hand side of an addition expression
; throws an error if e is not an addition expression
(define (addition-rhs e) ...)
```

Algebraic data type - example

- Interface of the algebraic data type can be used in the program
  - Independent of representation

```
(define (calc e)
   (cond [(addition? e) (+ (calc (addition-lhs e))
                           (calc (addition-rhs e)))]
         [(literal? e) (literal-value e)]))

> (make-addition
     (make-addition (make-literal 0) (make-literal 1))
     (make-literal 2))
'(addition (addition (literal 0) (literal 1)) (literal 2))

> (calc (make-addition
     (make-addition (make-literal 0) (make-literal 1))
     (make-literal 2)))
3
```

Representation, e.g: S-expression

Algebraic data types with lists and S-expressions

- S-Expressions
  - Nested lists
  - Universal data structure

```
(define (make-literal n)
  (list 'literal n))
(define (literal-value l)
  (if (literal? l)
    (second l)
    (error 'not-a-literal)))
(define (literal? l)
  (and
    (cons? l)
    (symbol? (first l))
    (symbol=? (first l) 'literal)))
...
```

All constructors, predicates and selectors must be implemented by the developer

Algebraic data types with structure definitions

- The define-struct construct automatically generates functions
  - Constructors
  - Selectors
  - Predicates

- In the example: Functions in the interface match the naming conventions of structs
  - Language takes the effort out of implementation

```
(define-struct literal (value))
(define-struct addition (lhs rhs))
```

  - Function calc is independent of the data representation and works unchanged

Algebraic data types with define-type

- Structure definitions for product types only
- However, algebraic data types have alternatives (sum type)

- define-type construct
  - Alternatives of product types
  - For each alternative
    - Fields
    - Together with predicate: Which values are permitted
  - Automatic generation of constructors, selectors, predicates
  - Predicates of fields become dynamic contracts
  - (Original implementation of calc works unchanged)

```
#lang racket
(require 2htdp/abstraction)
```

Algebraic data types with define-type

```
(define-type expression
  (literal (value number?))
  (addition (left Expression?) (right Expression?)))

> (make-addition
    (make-addition (make-literal 0) (make-literal 1))
    (make-literal 2))
(addition (addition (literal 0) (literal 1)) (literal 2))
```

Algebraic data types with define-type

- Additional (type) tests compared to define-struct
  - Field values must fulfill predicate

```
> (make-addition 0 1)
make-addition: contract violation
  expected: (or/c undefined? Expression?)
  given: 0
  in: the 1st argument of
      (->
       (or/c undefined? Expression?)
       (or/c undefined? Expression?)
       addition?)
  contract from: make-addition
  blaming: use
   (assuming the contract is correct)
    at: eval:2:14
```

Algebraic data types with define-type

- Additional (type) tests compared to define-struct
  - Field va

or/c means: one of the following contracts.

```
> (make-additi      )
make-addition: contract violation
  expected: (or/c undefined? Expression?)
  given: 0
  in: the 1st argument of
      (->
       (or/c undefined? Expression?)
       (or/c undefined? Expression?)
       addition?)
  contract from: make-addition
  blaming: use
   (assuming the contract is correct)
    at: eval:2:14
```

undefined? is the contract for a value that fulfills every type.

Algebraic data types with define-type

- Extension to pattern matching: type-case
  - Specification of the type for the value that is "matched"
  - Differentiation between the alternatives

- Enables static type testing
  - Are all alternatives of the matched type covered?

Algebraic data types with define-type

```
(define (calc e)
   (type-case expression e
      [literal (value) value]
      [addition (e1 e2) (+ (calc e1) (calc e2))]))
> (calc (make-addition
      (make-addition (make-literal 0) (make-literal 1))
      (make-literal 2)))
3

> (define (calc2 e)
      (type-case expression e
         [addition (e1 e2) (- (calc2 e1) (calc2 e2))]))
type-case: syntax error; probable cause: you did not include
a case for the literal variant, or no else-branch was present
```

Algebraic data types with define-type

- type-case
  - Completeness check
  - But: Limitation of supported patterns
    - Variant name and fields
    - No literals, nested patterns, etc.

$name_{i\_1}$ … $name_{i\_ni}$ may be used in *body-expression* $._i$

- General form

(type-case *type e*
  [*$variant_1$* (*$name_{1\_1}$* … *$name_{1\_n1}$*) *body-expression*]$_1$
  …
  [*$variant_m$* (*$name_{m\_1}$* … *$name_{m\_nm}$*) *body-expression$_m$*])