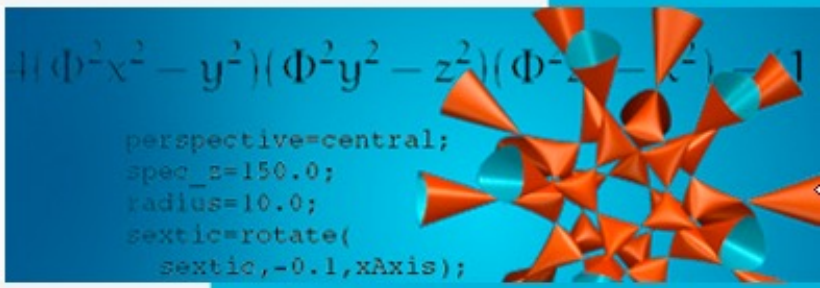
Philipps Universität Marburg

Declarative programming

Summer semester 2024

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

Imke Gürtler, Daniel Hinkelmann, Aaron Schafberg, Stefan Störmer



[Script 8]

Possible program results

- Meaning of a program: Sequence of reduction steps
- Possible results
  - Reduction to one value
  - Abort with error
  - Non-terminating sequence
- Values:
  - All constants (literals)
  - Instances of structures

Instances of structures

- Representation: '<' 'make-'<name> <v>* '>'

- Differentiation
  - Expression for generation with round brackets
    (make-posn 3 (+ 2 2))
  - Value of an instance with angle brackets
    <make-posn 3 4>

Surroundings

- Property of an expression in the program
- Contains known definitions
- Depending on the position of the expression
- Environment of an expression contains all definitions that precede the currently evaluated expression

Display of the surroundings

- The structure of the environment can also be defined by grammar

<env> ::= <env-element>*

<env-element> ::= '(' 'define' '(' <name> <name>+ ')' <e> ')'
| '(' 'define' <name> <v> ')'
| '(' 'define-struct' <name> '(' <name>+ ')' ')'

Display of the surroundings

- The structure of the environment can also be defined by grammar

<env> ::= <env-element>*

<env-element> ::= '(' 'define' '(' <name> <name>+ ')' <e> ')'
| '(' 'define' <name> <v> ')'
| '(' 'define-struct' <name> '(' <name>+ ')' ')'

Note: Constants are defined by a **value** in the environment.
In contrast, they are defined by an **expression** in the BSL language.

Repetition

- Evaluation items
- Congruence rule

Repetition

- Evaluation items
  - Marks sub-expressions that may be evaluated next
- Congruence rule
  - If e has a sub-expression e_2 in an evaluation item with $e_2 \rightarrow e_2'$, then $e \rightarrow e'$ applies, where e' is generated from e by replacing e_2 with e_2'.
  - So: if an expression has sub-expressions, these can be evaluated and the sub-expressions can be replaced by their values.
  - (* (+ 1 2) (+ 3 4)) → Congruence rule → (* 3 (+ 3 4))

Evaluation position

Evaluation position

Evaluation context

- Formalization of congruence rule and evaluation position
- Grammar with "hole": '[]'
- Each element contains exactly one hole

Evaluation context

```
<E> ::= '[]'
      | '(' <name> <v>* <E> <e>* ')'
      | '(' 'cond' '[' <E> <e> ']' {'[' <e> <e> ']'}* ')'
      | '(' 'and' <E> <e>+ ')'
      | '(' 'and' 'true' <E> ')'
```

Hole as an alternative.

To the left of the hole are only values: Evaluation from left to right.

Use <E> for evaluation item, <e> for non-evaluation items.

Mirrors production <e>, but adds "holes".

Evaluation context

- Which of these are evaluation contexts?

1. (posn-x (make-posn 14 []))
2. (and true (and [] x))
3. (make-posn 14 17)
4. (and x [])

Evaluation context

- Which of these are evaluation contexts?

1. (posn-x (make-posn 14 []))
2. (and true (and [] x))
3. ~~(make-posn 14 17)~~
4. ~~(and x [])~~

No hole

Hole in the wrong position

Evaluation sequence

- Previously: Evaluation items can be evaluated in any order

- Now: Evaluation sequence from left to right

Evaluation items

- Expressions can be used in an evaluation context
- E[e]
  - Replace hole in evaluation context with expression e
- Example
  - E = (* [] (+ 3 4))
  - E[(+ 1 2)] = (* (+ 1 2) (+ 3 4)).

Congruence rule with evaluation context

- **(KONG):** If $\mathbf{e_1 \rightarrow e_2}$, then $\mathbf{E[e_1] \rightarrow E[e_2]}$.

- Example
  - e = **(* (+ 1 2) (+ 3 4))**
  - E = **(* [] (+ 3 4))**
  - e_1 = **(+ 1 2)** with e = E[e]$_1$
  - Reduction of e_1 **: (+ 1 2)** → **3**
  - Therefore reduction of e: e → E[3] = (* 3 (+ 3 4))

Naming conventions

- Identifiers in rule definitions
  - Non-terminal <x>
  - Then variants of x stand for any words of the non-terminal (x_1 , x_2 , x, x')
- These identifiers are "meta variables"
  - Meta: Variable not via values in the program, but via program parts

Quantity notation

- Non-terminal as a set:
  Set of all words that can be derived from non-terminals

- Set notation for congruence rule

  For all $e_1 \in$<e> and all $e_2 \in$<e> and all E $\in$ <E>

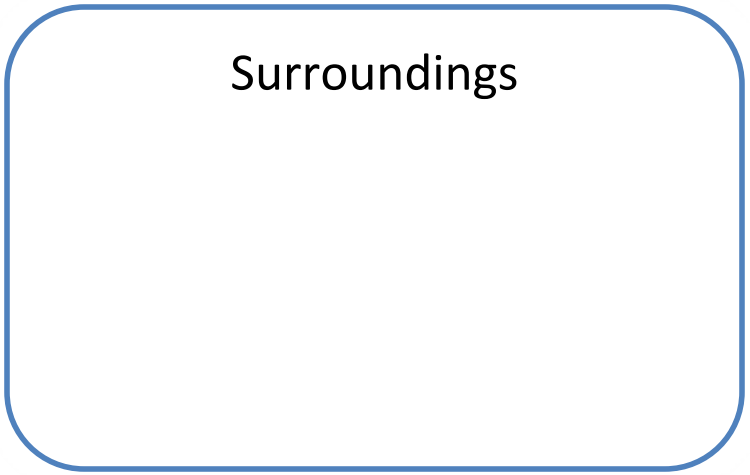
  If $e_1 \rightarrow e_2$, then $E[e_1] \rightarrow E[e]_2$

Importance of programs

- **(PROG):** A program is executed from left to right and starts with the empty environment. If the next program element is ...
  - ... a function or structure definition, this definition is included in the environment and execution is continued with the next program element in the extended environment.
  - ... an expression, it is evaluated to a value in the current environment according to the following rules.
  - ... a constant definition (define x e), e is first evaluated to a value v in the current environment and then (define x v) is added to the current environment.

Example

```
(define (f x) (+ x 1))
(define c (f 5))
(+ c 3)
```



Example

```
(define (f x) (+ x 1))
(define c (f 5))
(+ c 3)
```

Surroundings

```
(define (f x) (+ x 1))
```

Example

```
(define (f x) (+ x 1))
(define c (f 5))
(+ c 3)
```

In environment "(define (f x) (+ x 1))" evaluate to 6.

Surroundings

```
(define (f x) (+ x 1))
(define c 6)
```

Example

```
(define (f x) (+ x 1))
(define c (f 5))
(+ c 3)
```

Evaluate in current environment to 9.

Surroundings

```
(define (f x) (+ x 1))
(define c 6)
```

Evaluation rules

- Notation: Reduction rules
- Evaluation of a printout in the environment
  - Environment implicitly given with reduction rule
- Expression e is evaluated by reducing it to a value:
  $e \rightarrow e_1 \rightarrow ... \rightarrow v$
- Expression no value, but no reduction rule applicable: Program error

Reduction rules for function call

- Primitive functions
  - Not defined in environment
  - Evaluation built in
  - **(PRIM):** If ***name*** is a primitive function ***f*** and $f(v_1,...,v_n)=v$, then $(name\ v_1 \dots v_n) \rightarrow v$
- User-defined function
  - Definition contained in environment
  - Evaluation by body expression
  - **(FUN):** If (define (***name*** $name_1 \dots name_n$) ***e***) in environment, then $(name\ v_1 \dots v_n) \rightarrow e[name_1 := v_1 \dots name_n := v]_n$

Replace occurrences of formal parameters in *e* with the passed argument values.

Reduction rule for constants

- **(CONST):** If (define ***name v***) in the environment, then ***name → v***

Reduction rules for conditional expressions

- Always evaluate the first condition expression according to the evaluation context

- **(COND-True):** (cond [true ***e***] ...) → ***e***
- **(COND-False):** (cond [false e_1] [e_2 e_3] ...) →
  (cond [e_2 e_3] ...)

As a reminder, the evaluation context:

```
<E> ::= '[]'
        | '(' <name> <v>* <E> <e>* ')'
        | '(' 'cond' '[' <E> <e> ']' {'[' <e> <e> ']'}* ')'
        | '(' 'and' <E> <e>+ ')'
        | '(' 'and' 'true' <E> ')'
```

Reduction rules for Boolean expressions

- If one of the operands is false, the entire expression is false
  - Evaluation up to the first false

- **(AND-1):** (and true true) → true
- **(AND-2):** (and true false) → false
- **(AND-3):** (and false ...) → false
- **(AND-4):** (and true e_1 e_2 ...) → (and e_1 e_2 ...)

To ensure that a truth value is returned, both operands must be considered.

Reduction rules for structures

- Structure definition generated
  - Constructor (*make-name*)
  - Selectors (*name-field*)
  - Predicates (*name*?)

Reduction rule for constructor calls

- A constructor can be called if a corresponding struct is defined in the environment.
- Number of fields must match the number of arguments

- **(STRUCT-make):**
  If (define-struct ***name*** (***name*** $_1$ ... ***name*** $_n$)) in the environment, then
  (***make-name*** v_1 ... v_n) → < ***make-name*** v_1 ... v_n >.

Reduction rule for selector calls

- Structure definition maps field names to position within structure.

- (define-struct vel (deltax deltay))
- (make-vel 1 2) → <make-vel 1 2>
- (vel-deltax <make-vel 1 2>) → 1

deltax is in position 1,
deltay is in position 2

- **(STRUCT-select):**
  If (define-struct ***name*** (***name*** $_1$ ... ***name*** $_n$)) in the environment, then
  (***name-name*** $_i$ < ***make-name*** v_1 ... v_n >) → v_i

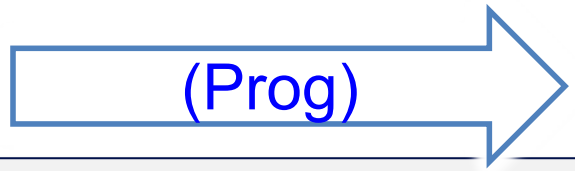
Reduction rules for structural predicates

- **(STRUCT-predtrue):**
  (***name***? < ***make-name*** ... >) → true

- **(STRUCT-predfalse):**
  If ***v*** is not < ***make-name*** ... >, then (***name***? ***v***) → false

Reduction using the example of

```
(define-struct s (x y))
(define (f x) (cond [(< x 1) (/ x 0)]
                    [true (+ x 1)]
                    [true x]))
(define c (make-s 5 (+ (* 2 3) 4)))
(f (s-x c))
```



Reduction using the example of

```
(define-struct s (x y))
(define (f x) (cond [(< x 1) (/ x 0)]
                    [true (+ x 1)]
                    [true x]))
(define c (make-s 5 (+ (* 2 3) 4)))
(f (s-x c))
```

Surroundings

(Prog)

Reduction using the example of

```
(define (f x) (cond [(< x 1) (/ x 0)]
                    [true (+ x 1)]
                    [true x]))
(define c (make-s 5 (+ (* 2 3) 4)))
(f (s-x c))
```

Surroundings

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

(Prog)

Reduction using the example of

```
(define c (make-s 5 (+ (* 2 3) 4)))
(f (s-x c))
```

Surroundings

```
(define-struct s (x y))
(define (f x) (cond [(< x 1) (/ x 0)]
                    [true (+ x 1)]
                    [true x]))
```

(Prog)

Reduction using the example of

```
(define c (make-s 5 (+ (* 2 3) 4)))
(f (s-x c))
```

e = (make-s 5 (+ (* 2 3) 4))
E = (make-s 5 (+ [] 4)) and e_1 = (* 2 3)
(PRIM) e_1 → 6
(KONG) e → (make-s 5 (+ 6 4))

Surroundings

```
(define-struct s (x y))
(define (f x) (cond [(< x 1) (/ x 0)]
                    [true (+ x 1)]
                    [true x]))
```

(Prog)

Reduction using the example of

```
(define c (make-s 5 (+ 6 4)))
(f (s-x c))
```

e = (make-s 5 (+ 6 4))
E = (make-s 5 []) and e_1 = (+ 6 4)
(PRIM) e_1 → 10
(KONG) e → (make-s 5 10)

Surroundings

```
(define-struct s (x y))
(define (f x) (cond [(< x 1) (/ x 0)]
                    [true (+ x 1)]
                    [true x]))
```

(Prog)

Reduction using the example of

```
(define c (make-s 5 10))
(f (s-x c))
```

(STRUCT-make)
(make-s 5 10) → <make-s 5 10>

Surroundings

```
(define-struct s (x y))
(define (f x) (cond [(< x 1) (/ x 0)]
                    [true (+ x 1)]
                    [true x]))
```

(Prog)

Reduction using the example of

(f (s-x c))

Surroundings

```
(define-struct s (x y))
(define (f x) (cond [(< x 1) (/ x 0)]
                    [true (+ x 1)]
                    [true x]))
(define c <make-s 5 10>)
```

(Prog)

Reduction using the example of

(f (s-x c))

e = (f (s-x c))
E = (f (s-x [])) and e_1 = c
(CONST) c → <make-s 5 10>
(KONG) e → (f (s-x <make-s 5 10>))

Surroundings

```
(define-struct s (x y))
(define (f x) (cond [(< x 1) (/ x 0)]
                    [true (+ x 1)]
                    [true x]))
(define c <make-s 5 10>)
```

(Prog)

Reduction using the example of

(f (s-x <make-s 5 10>))

e = (f (s-x <make-s 5 10>))
E = (f []) and e_1 = (s-x <make-s 5 10>)
(STRUCT-select) e_1 → 5
(KONG) e → (f 5)

Surroundings

```
(define-struct s (x y))
(define (f x) (cond [(< x 1) (/ x 0)]
                    [true (+ x 1)]
                    [true x]))
(define c <make-s 5 10>)
```

(Prog)

Reduction using the example of

(f 5)

e = (f 5)
(FUN) e → (cond [(< 5 1) (/ 5 0)]
[true (+ 5 1)]
[true 5])

Surroundings

```
(define-struct s (x y))
(define (f x) (cond [(< x 1) (/ x 0)]
                    [true (+ x 1)]
                    [true x]))
(define c <make-s 5 10>)
```

(Prog)

Reduction using the example of

```
(cond [(< 5 1) (/ 5 0)]
      [true (+ 5 1)]
      [true 5])
```

Surroundings

```
(define-struct s (x y))
(define (f x) (cond [(< x 1) (/ x 0)]
                    [true (+ x 1)]
                    [true x]))
(e-s 5 10>)
```

e = (cond [(< 5 1) (/ 5 0)] [true (+ 5 1)] [true 5])
E = (cond [[] (/ 5 0)] [true (+ 5 1)] [true 5])
and e_1 = (< 5 1)
(PRIM) e_1 → false
(KONG) e → (cond [false (/ 5 0)] [true (+ 5 1)] [true 5])

(Prog)

Reduction using the example of

(cond [false (/ 5 0)]
[true (+ 5 1)]
[true 5])

e = (cond [false (/ 5 0)] [true (+ 5 1)] [true 5])
(COND-False) e → (cond [true (+ 5 1)] [true 5])

Surroundings
(define-struct s (x y))
(define (f x) (cond [(< x 1) (/ x 0)]
[true (+ x 1)]
[true x]))
c <make-s 5 10>)

(Prog)

Reduction using the example of

(cond [true (+ 5 1)]
[true 5])

Surroundings
(define-struct s (x y))
(define (f x) (cond [(< x 1) (/ x 0)]
[true (+ x 1)]
[true x]))
c <make-s 5 10>)

e = (cond [true (+ 5 1)] [true 5])
(COND-True) e → (+ 5 1)

(Prog)

Reduction using the example of

(+ 5 1)

e = (+ 5 1)
(PRIM) e → 6

Surroundings
(define-struct s (x y))
(define (f x) (cond [(< x 1) (/ x 0)]
[true (+ x 1)]
[true x]))
c <make-s 5 10>)

(Prog)

Reduction using the example of

6

Surroundings

```
(define-struct s (x y))
(define (f x) (cond [(< x 1) (/ x 0)]
                    [true (+ x 1)]
                    [true x]))
(define c <make-s 5 10>)
```