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 13, 12]

Language level ISL+

- Intermediate Student Language Plus (ISL+)
  - Everything we have come to know so far
- Extends BSL with some constructs
  - Local definitions
  - View functions as values (with environment) (closures)

Meaning of ISL+

- Definition of the meaning as for BSL
  - Core language
  - Syntax as EBNF grammar
  - Semantics according to the grammar rules
  - Evaluation rules
  - Reduction ratio
- No redefinition of meaning for language concepts that are not affected by new concepts
  - Structure definitions (define-struct)
  - Conditional expressions (cond)
  - Logical operators (and)
- Simplification through uniform treatment of functions and values
  - Function definition is syntactic sugar for definition of a constant with lambda expression

Syntax of the core language

```
<definition> ::= '(' 'define' <name> <e> ')'
<e> ::= '(' <e> <e>+ ')'
      | '(' 'local' '[' <definition>+ ']' <e> ')'
      <name>
      | <v>
<v> ::= '(' 'lambda' '(' <name>+ ')' <e> ')'
      <number>
      <boolean>
      <string>
      <image>
      | '+'
      | '*'
      | ...
```

Names of all primitive functions

Surroundings

- By eliminating function definitions (and omitting structure definitions), the **environment** is simplified

- Only sequence of constant definitions

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

<env-element> ::= '(' 'define' <name> <v> ')'

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 functi … n is included in the environme … e next program element ir~~

    Identical to the previous definition except for the omission of the "structure or function definition" case

  - ... an expr … e 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.

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 funct... r structure definition, this definition is included in the environme... ecution is continued with the next program elem...~~
  - ~~... ar... urrent envi...~~
  - ... 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.

The evaluation rule defined later (LOCAL) influences what the "next program element" is.

Evaluation positions and the congruence rule

- In contrast to BSL, an expression also occurs in the first position of a function call

<E> ::= '[]'
 | '(' <v>* <E> <e>* ')'

The function name was in BSL here. A closure value can now be placed in the first position.

- The congruence rule applies unchanged
  - (KONG): If $e_1 \rightarrow e_2$, then $E[e_1] \rightarrow E[e_2]$.

Philipps Universität Marburg

Meaning of function calls

- There are no more function definitions
  - Instead: Constants with lambda as value
  - Constant name that stands for the function is evaluated to lambda expression
  - Instead of a function name: Lambda expression in first position

- Therefore:
  - Elimination of the rule (FUN) for function calls
  - New: Rule (APP) for application of a lambda expression

Meaning of function calls

- **(APP):** ((lambda (***name***$_1$ … ***name***$_n$) ***e***) ***v***$_1$ … ***v***$_n$) → **e[*name*$_1$:= *v*$_1$ … *name*$_n$:= *v*]$_n$**
- Replacement of formal parameters must comply with scoping rules
  - Not all occurrences of the identifier can simply be replaced
  - Example:
    - ((lambda (x) (+ x 1)) (* x 2))[x := 7]

      = ((lambda (x) (+ x 1)) (* 7 2)))

These are different variables with the same name, but in different scopes.

In this case, only the x in the outer scope may be replaced.

Meaning of function calls

- If the name of a primitive function is in the first position of an expression, (PRIM) applies analogously to the previous definition
  - **(PRIM):** If v is a primitive function **f** and $\mathbf{f}(v_1, \ldots, v_n) = v'$, then (v v_1 ... v_n) → v'.

A value that corresponds to a primitive function is shown here instead of the function name as before.

Primitive functions can also be the result of calculations, e.g: ((if <cond> + *) 3 4)

Importance of local definitions

- Difference to global definitions
  - Limited scope of validity
  - Access to local context

- Evaluation is dynamic!
  - During evaluation, they can be treated similarly to global definitions

- Differences:
  - Scope plays a role in the search for a definition
  - Local context is accessible during evaluation

Importance of local definitions

- For the evaluation:
  - Replace local definition with global definition
  - Access to local context has already been replaced by rules such as (APP) or (CONST) ("substitution")

- (LOCAL):
  E[(local [(define $name_1$ e_1) ... (define $name_n$ e_n)] e)]
  → (define $name_1{}'$ $e_1{}'$) ... (define $name_n{}'$ $e_n{}'$) E[e']
  where $name_1{}'$, ..., $name_n{}'$ are "fresh" names that do not appear anywhere else in the program
  and e', $e_1{}'$, ..., $e_n{}'$ are copies of e, e_1 , ..., $e_n{}'$ in which all occurrences of $name_1$, ..., $name_n$ are replaced by $name_1{}'$, ..., $name_n{}'$.

Importance of local definitions

- For the evaluation:
  - Replace local definition with global definition
  - Access to local context has already been replaced by rules such as (APP) or (CONST) ("substitution")

- (LOCAL):
  E[(local [(define $name_1$ e_1) ... (define $name_n$ e_n)] e)]
  → (define $name_1$ ' e_1 ') ... (define $name_n$ ' e_n ') E[e']
  where nar... ' ... $name_n$ ' are "fresh" names that do not
  appea...
  and e... hich all
  occur... $name_1$
  ', ..., ...

The local definitions are treated like global definitions and become the "next program element" for the evaluation according to (PROG). Once the definitions have been processed, the evaluation E[e'] is continued.

Importance of local definitions

- For the evaluation:
  - Replace local definition with global definition
  - Access to local context has already been replaced by rule (APP) ("substitution")

Names are replaced so that they are only used within the scope of the local expression.

- (LOCA
  E[(loca ne $name_n$ e_n)] e)]
  → (define $name_1$ ' e_1 ') ... (define $name_n$ ' e_n ') E[e']
  where $name_1$ ', ..., $name_n$ ' are "fresh" names that do not appear anywhere else in the program
  and e', e_1 ', ..., e_n ' are copies of e, e_1 , ..., e_n ' in which all occurrences of $name_1$, ..., $name_n$ are replaced by $name_1$ ', ..., $name_n$ '.

Importance of local definitions

- For the evaluation:
  - Replace local definition with global definition
  - Access to local context has already been replaced by rule (APP) ("substitution")

Accesses to formal parameters in e', e_1 ', ... e_n ' have already been replaced by previous evaluation (APP). Access to local definitions is possible as these are now in the global environment.

- (LOCA
  E[(loc … *e*)]
  → (de …]
  where … not
  appear anyw… else in the program
  and *e'*, *e_1 '*, ..., *e_n '* are copies of *e*, *e_1* , ..., *e_n '* in which all occurrences of *$name_1$* , ..., *$name_n$* are replaced by *$name_1$ '*, ..., *$name_n$ '*.

Importance of local definitions

```
(f 2)
→ (KONG, CONST, APP)
(+ 2
  (local
    [(define y (+ 2 1))]
    (* y 2)))
→ (LOCAL)
(define y_0 (+ 2 1))
(+ 2 (* y_0 2))
→ (PROG, PRIM)
(+ 2 (* y_0 2))
→ (KONG, CONST)
(+ 2 (* 3 2))
→* (PRIM)
8
```

```
(define f (lambda (x)
  (+ 2
    (local
      [(define y (+ x 1))]
      (* y 2)))))
```

For rule (APP): Substitution of the formal parameters with arguments.

New in the area:
(define y_0 3)

Importance of local definitions

```
(+ 2 (local
   [(define THREE 3)]
   (local
      [(define y (+ THREE 1))]
      (* y 2))))
→ (LOCAL)
(define THREE_0 3)
(+ 2 (local
      [(define y (+ THREE_0 1))]
      (* y 2)))
→ (PROG)
(+ 2 (local
      [(define y (+ THREE_0 1))]
      (* y 2)))
→ (LOCAL)
(define y_0 (+ THREE_0 1))
(+ 2 (* y_0 2))
→ (PROG, CONST, PRIM, KONG)
10
```

Surroundings:
(define THREE_0 3)

THREE_0 is in the evaluation environment

Surroundings:
(define THREE_0 3)
(define y_0 4)

Scoping in evaluation rules

- Semantics: lexical scoping
  - Scope of validity of local definitions
  - Only within (local ...) expression

- Scoping rule is implemented by two evaluation rules
  - (LOCAL]
  - (APP)

Scoping in evaluation rules

- (LOCAL]
  - Renaming of local constants
  - New name must not be defined anywhere else in the program
  - Within the (local ...) expression
    - Replace the original name
    - Through new name

- Generated name is different from all user-defined names
  - Cannot be used anywhere by chance
  - Compiler prohibits the use of constants without definition in the scope
- Insert the generated name only within the (local ...) expression

Scoping in evaluation rules

- (APP)
  - Similar to (LOCAL):
    - Replacement of the formal parameters in the function body
    - Replacement of "outside" with "inside"

- For closures as a result of the (APP) evaluation rule
  - Bound parameters are already replaced
  - Example

```
(define (f x)
  (lambda (y) (+ x y)))
(f 3)
→ (FUN)
(lambda (y) (+ 3 y))
```

x is bound to the formal parameter.

x is replaced by the argument.

Shadowing

- When using a name (constant or formal parameter)
  - Binding to definition
  - There can be several definitions for the name
    - In the current scope
    - In the surrounding scopes
    - Only one definition per scope
  - Binding always to the lexically closest definition

Shadowing

- Definition of a constant or a formal parameter: "binding occurrence" of the identifier

- Use of an identifier as an expression: "bound occurrence"

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

Binding occurrence of x (constant)

Binding occurrence of x (formal parameter)

Bound occurrence of x

Binding occurrence of x (constant)

Bound occurrence of x

Shadowing

- Definition of a constant or a formal parameter: "binding occurrence" of the identifier

- Use of an identifier as an expression: "bound occurrence"

```
(define x 1)
(define (f x)
  (+ x (local [(define x 2)] (+ x 1))))
> (f 3)
6
```

Shadowing and modularity

- Module
  - Independently understandable program unit
  - The meaning of the module should be the same, regardless of the context in which it is located
- Example: Printout as a module
  - Use of unbound identifiers: no definition for identifiers in the current scope
  - Use of bound identifiers: Definition for identifiers in the current scope
  - Expressions without unbound identifiers: "**closed term**"
- The meaning of a closed term is independent of the place where it is defined

Scoping Modularity

- Programming language concept: "block structure"
  - Local definition of identifiers
  - Lexical scoping
  - Shadowing

- First programming language with block structure: ALGOL 60

- Because a block structure is good for program comprehension, it is used by most modern programming languages

Pattern Matching

- Simplifies case distinctions
  - Declarative description of the cases
  - Naming the structural elements for further use

- Examples

```
(define (f x)
  (match x
    [7 8]
    [else 9]))
>(f 7)
8
```

"Match" depending on the form of x

x is the literal 7, then the result is 8

Otherwise the result is 9

Switch language to "Advanced"!

Pattern Matching

- Examples

```
(define (f x)
  (match x
    [(list 1 y 3) y]))
```

Placeholders stand for any values and are bound.

Bound placeholders can be used in the result printout.

Not only literals, but also compound values can be specified as a condition.

```
>(f (list 1 2 3))
2
```

Pattern Matching

- Examples

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

(define (f x)
  (match x
    [(struct point (y y)) y]))
```

Placeholders can occur multiple times in the pattern. The pattern then only matches if the same value appears in all positions.

Binding the values eliminates the need for selectors.

Pattern Matching

- Examples

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

(define (f x)
  (match x
    [(cons (struct point (1 z)) y) z]))
```

Data structures that are matched to can also be nested as deeply as required.

Pattern Matching

- Examples

```
(define (f x)
  (match x
    [7 8]
    ["hey" "joe"]
    [(list 1 y 3) y]
    [(cons a (list 5 6)) (add1 a)]
    [(struct point (5 5)) 42]
    [(struct point (y y)) y]
    [(posn y z) (+ y z)]
    [(cons (struct point (1 z)) y) z]))
```

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

Any number of match clauses

Only the result printout of the first successful match is evaluated. For example:

```
>(f (make-point 5 5))
42
```

If no expression matches, an error message appears:

match: no matching clause for ...

Analog to cond.

Pattern Matching

- Examples

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

```
(define (f x)
  (match x
    [7 8]
    ["hey" "joe"]
    [(list 1 y 3) y]
    [(cons a (list 5 6)) (add1 a)]
    [(struct point (5 5)) 42]
    [(struct point (y y)) y]
    [(posn y z) (+ y z)]
    [(cons (struct point (1 z)) y) z]))
```

For (built-in) lists.

For the built-in structure posn:
(f (make-posn 5 7))
12

Philipps Universität Marburg

Pattern Matching

- Instead of:

```
(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]))
```

- With pattern matching:

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

Meaning of pattern matching

- (match ...) is an expression and can be used wherever an expression is expected

- General form:

(match v [(p_1 e_1)] ... [(p_n e_n)])

- Meaning informal:
  - Find the first p_i that matches v
  - Bind the variables that occur in p_i with values from v
  - Replace the occurrences of the variable e_i with the bound values

Meaning of pattern matching

- Let's look at matching as a function
  - Input: Pattern and value
  - Output: "no match" or substitution

- Substitution:
  - $[x_1 := v_1 , \ldots, x_n := v]_n$
  - x_i : Identifier
  - v_i : Values

Meaning of pattern matching

- match(*v*, *v*) = [] (the empty substitution)
- match(*x*, *v*) = [*x* := *v*]
- match(**(struct *id* (*p*$_1$ ... *p*$_n$))** , **(*make-id v*$_1$ ... *v*$_n$**)) = match(***p*$_1$, *v*$_1$**) + ... + match(***p*$_n$, *v*)$_n$**
- Special case for lists and built-in structures analog
- match(...,...) = "no match" in all other cases

Operator "+": Combination of substitutions.

Meaning of pattern matching

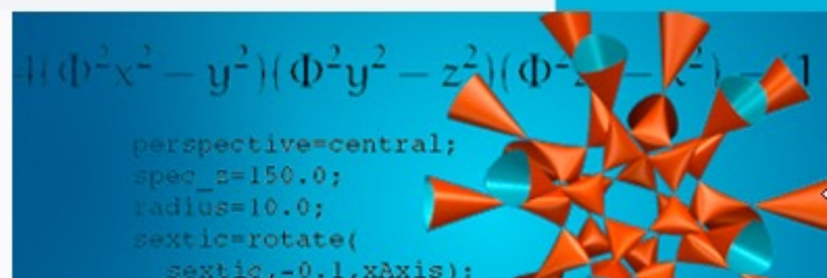
- Combination of substitutions
  - One of the two substitutions is "no match" → "no match"
  - Both substitutions contain mapping for the same name, but with different values → "no match"
  - Otherwise union of the mappings from both substitutions

Meaning of pattern matching

- If an expression e has the form (match v [(p_1 e_1)] ... [(p_n e_n)])
- If match(p_1 , v) = [x_1 := v_1 , ..., x_n := v $]_n$
  - Then the following applies: $e \rightarrow e_1$ [x_1 := v_1 , ..., x_n := v $]_n$
- If match(p_1 , v) = "no match"
  - Dan applies: $e \rightarrow$ (match v [(p_2 e_2)] ... [(p_n e_n)])

Philipps Universität Marburg

Repetition
Proof of equivalence with induction



Repetition: Proof of equivalence by induction

- Recursive calculation of the sum of the numbers from 1 to n
  - ```
    (define (sum n)
          (cond [(zero?  n)  0]
                      [else (+  n (sum (-  n 1)))]))
    ```
- Gaussian summation formula
  - $\sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}$
- Therefore, show that applies:
  - (sum n) ≡ (/ (* n (+ n 1)) 2)
    ```

Repetition: Proof of equivalence by induction

```
(define (sum n)
    (cond [(zero? n) 0]
          [else (+ n (sum (- n 1)))]))
```

- To show:
  - (sum n) ≡ (/ (* n (+ n 1)) 2)
- Base case (1)
  - n = 0
  - (sum 0)
  - EFUN ≡ EFUN
  - (cond [(zero? 0) 0] [else (+ 0 (sum (- 0 1)))]))
  - ≡ EKONG with ERED with STRUCT-predtrue
  - (cond [true 0] [else (+ 0 (sum (- 0 1)))]))
  - ≡ ERED with COND-true
  - 0

Repetition: Proof of equivalence by induction

- To show:
  - (sum n) ≡ (/ (* n (+ n 1)) 2)
- Base case (2)
  - n = 0
  - (/ (* 0 (+ 0 1)) 2)
  - ≡ PRIM
  - 0
  - According to ETRANS and EKOMM is therefore (sum 0) ≡ (/ (* 0 (+ 0 1)) 2)

```
(define (sum n)
  (cond [(zero? n) 0]
        [else (+ n (sum (- n 1)))]))
```

Repetition: Proof of equivalence by induction

- To show:
  - (sum n) ≡ (/ (* n (+ n 1)) 2)
- Induction step:
  n = (add1 n')
  - We may use:
  - (sum n') ≡ (/ (* n' (+ n' 1)) 2)

```
(define (sum n)
   (cond [(zero? n) 0]
            [else (+ n (sum (- n 1)))]))
```

Repetition: Proof of equivalence by induction

- To show:
  - (sum n) ≡ (/ (∗ n (+ n 1)) 2)
- Induction step: n = (add1 n')
  - We may use:
  - (sum n') ≡ (/ (∗ n' (+ n' 1)) 2)

```
(define (sum n)
   (cond [(zero? n) 0]
         [else (+ n (sum (- n 1)))]))
```

```
n = (add1 n')
(sum n') ≡ (/ (∗ n' (+ n' 1)) 2)
```

- Proof (1)
  - (sum (add1 n'))
  - EFUN ≡ EFUN
  - (cond [(zero? (add1 n')) 0] [else (+ (add1 n') (sum (- (add1 n')) 1)))])

Starting from the left side of the equivalence.

Philipps Universität Marburg

Repetition: Proof of equivalence by induction

- Induction step:

```
(define (sum n)
    (cond [(zero?  n)  0]
          [else (+  n (sum (-  n 1)))]))
```

```
n = (add1 n')
(sum n') ≡ (/  (*  n'  (+  n'  1))  2)
```

- Proof (1)
  - ...
  - (sum (add1 n'))
  - EFUN ≡ EFUN
  - (cond [(zero? (add1 n')) 0] [else (+ (add1 n') (sum (- (add1 n')) 1)))])
  - ≡ EKONG with ERED with STRUCT-predfalse
  - (cond [false 0] ...)
  - ≡ ERED with COND-false
  - (cond [else (+ (add1 n') (sum (- (add1 n') 1)))])

Repetition: Proof of equivalence by induction

```
(define (sum n)
   (cond [(zero?  n)  0]
         [else (+  n (sum (-  n 1)))]))
```

```
n = (add1 n')
(sum n') ≡ (/  (*  n'  (+  n'  1))  2)
```

- Induction step:
- Proof (1)
  - ...
  - (cond [else (+ (add1 n') (sum (- (add1 n') 1)))])
  - ≡ ERED with COND-true
  - (+ (add1 n') (sum (- (add1 n') 1)))
  - ≡ EKONG with EPRIM
  - (+ (add1 n') (sum n'))
  - ≡ EKONG with induction acceptance
  - (+ (add1 n') (/ (* n' (+ n' 1)) 2))

Repetition: Proof of equivalence by induction

- Induction step:

Starting from the right-hand side of equivalence.

```
(define (sum n)
   (cond [(zero? n) 0]
         [else (+ n (sum (- n 1)))]))
```

```
n = (add1 n')
(sum n') ≡ (/ (* n' (+ n' 1)) 2)
```

- Proof (2)
  - (/ (* (add1 n') (+ (add1 n') 1)) 2)
  - ≡ EPRIM
  - (+ (add1 n') (/ (* n' (+ n' 1)) 2))

$$\frac{(n'+1)\cdot((n'+1)+1)}{2}$$
$=$
$=$
$=$
$=$

Repetition: Proof of equivalence by induction

- Induction step:

```
(define (sum n)
    (cond [(zero? n) 0]
          [else (+ n (sum (- n 1)))]))
```

- Proof

```
n' = (add1 n')
(sum n') ≡ (/ (* n' (+ n' 1)) 2)
```

  - According to ETRANS and EKOMM, therefore
  - (sum (add1 n')) ≡ (/ (* (add1 n') (+ (add1 n') 1)) 2)
  - And with n = (add1 n'): (sum n) ≡ (/ (* n (+ n 1)) 2)