λ-calculus formal verification

DCC-831

Augusto G. Lima

Department of Computer Science

December 2, 2025

Introduction

- Formalism for computations
- Two basic operations: Abstraction and Application
- Grammar confers **tree** structure
- $x \in V \Rightarrow x \in \Lambda$
  $M \in \Lambda, x \in V \Rightarrow (\lambda x.M) \in \Lambda$
  $M, N \in \Lambda \Rightarrow (MN) \in \Lambda$
- $(\rightarrow_\beta)$: β-reductions

Initial modeling

Signatures

```
abstract sig Expression {}
sig Name {}

sig Variable extends Expression {
    var name : one Name }

sig Abstraction extends Expression {
    var param : one Variable,
    var body  : one Expression }

sig Application extends Expression {
    var func : one Expression,
    var arg  : one Expression }
```

Initial modeling

Utils

```
fun derivations[e: Expression]: set Expression {
    (e.(Abstraction<:param) +
    e.(Abstraction<:body)+
    e.(Application<:func) +
    e.(Application<:arg) ) }

fun subtree[e: Expression]: set Expression {
    e.*({e1,e2: Expression |
        e2 in e1.derivations})}
```

Initial modeling

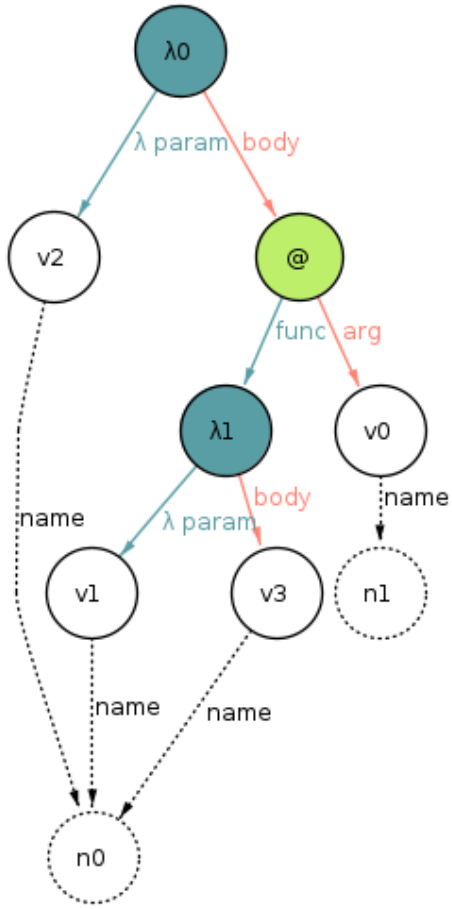
Grammar predicate

```
pred grammar_structure {
    one e: Expression |
        e not in Expression.derivations

    no e: Expression |
            e in e.^({e1,e2: Expression |
                e2 in e1.derivations})

    no e: Expression | e in e.derivations

    all ab: Abstraction | ab.param != ab.body
    all ap: Application | ap.func != ap.arg

    all e1,e2: Expression |
            e1!=e2 => no(e1.derivations
                & e2.derivations)}
```

Initial modeling

Discussion

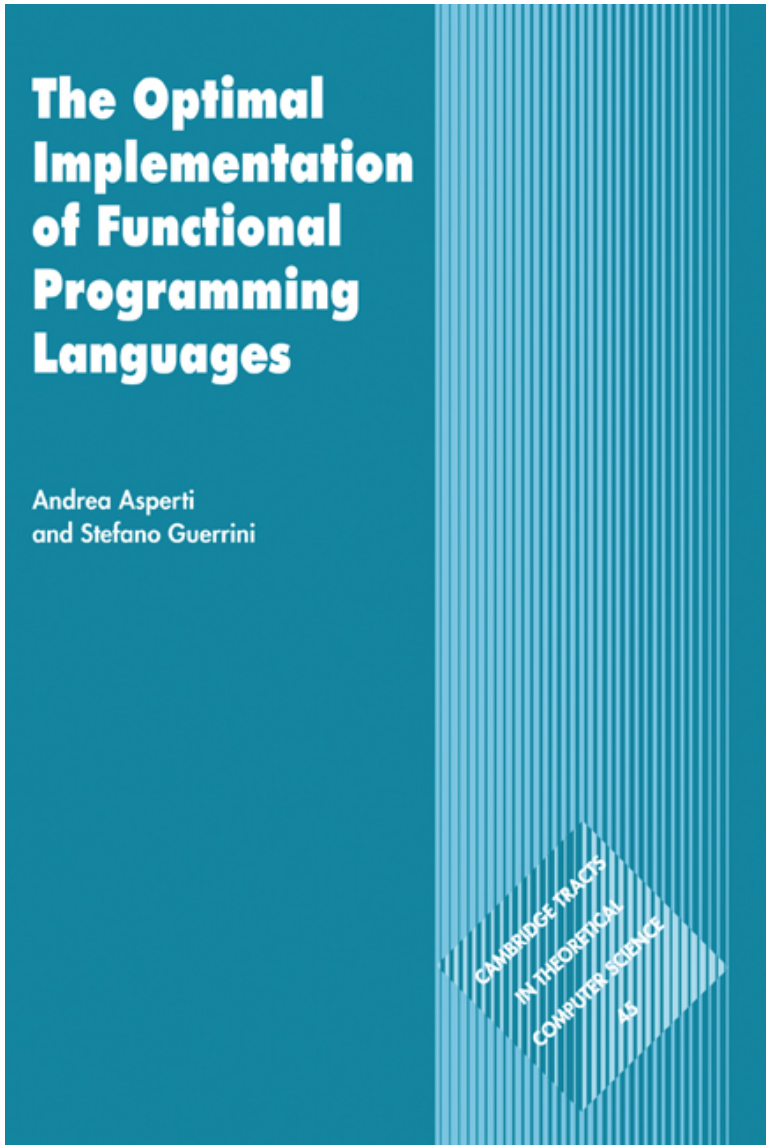
- Represents grammar rules
- Beautiful tree but too naive
- What is the issue? Can anyone notice it?
- Solution in next slide



Modeling

Sharing

- Heitor's recommendation
- Sharing: A system to share expressions
- Representation: from trees to DAGs
- Bound variables



[1]

Modeling

Signatures, dynamic system

```
abstract sig Status {}

one sig Active, Inactive extends Status {}

abstract sig Expression {
          var status : one Status}

abstract sig Reduction {}

one sig Alpha, Beta, None extends Reduction {}

one sig Track {
         var op: lone Reduction }
```

Modeling

Signatures, λ-calculus

```
sig Abstraction extends Expression {
        var param : one Variable,
        var body : one Expression
}

sig Application extends Expression {
        var func : one Expression,
        var arg : one Expression
}

sig Variable extends Expression {
        var binder: lone Abstraction
}
```

Modeling

New utils

```
fun subtree2[e: Expression, f: Expression]:
set Expression {
        e.*({e1,e2: Expression |
        (e2 in e1.derivations
        and e1 != f and e2 != f)})}

fun active_expressions [] : set Expression {
        {e: Expression | e.status = Active}
}

fun inactive_expressions [] : set Expression {
        {e: Expression | e.status = Inactive} }
```

Dynamic system

Initial well-formed expression

```
pred well_formed_expression {
no e: Expression | e.status = Inactive

one e: Expression |
    e not in Expression.derivations

no e: Expression |
       e in e.^({e1,e2: Expression |
       e2 in e1.derivations})

no e: Expression | e in e.derivations

                                        ...
```

Dynamic system

Initial well-formed expression

```
...

all a: Abstraction | a.param.binder = a

all v: Variable | some v.binder=>
((v=v.binder.param)
or (v in subtree[v.binder.body]))

all v1,v2: Variable |
(some v1.binder and some v2.binder and v1!=v2)=>
(v1.binder != v2.binder)

all v: Variable, e: Expression |
(e!=v and some v.binder and v in subtree[e])=>
(v not in subtree2[e,v.binder]
or e in subtree[v.binder])
}
```

Dynamic system

Frame conditions

```
pred noStatusChange [S : set Expression] {
        all e : S | e.status' = e.status}

pred noParamChange [S : set Abstraction] {
        all a : S | a.param' = a.param}

pred noBodyChange [S : set Abstraction] {
        all a : S | a.body' = a.body}

pred noFuncChange [S : set Application] {
        all a : S | a.func' = a.func}

pred noArgChange [S : set Application] {
        all a : S | a.arg' = a.arg}

pred noBinderChange [S : set Variable] {
        all v : S | v.binder' = v.binder}
```

Dynamic system

β-reduction

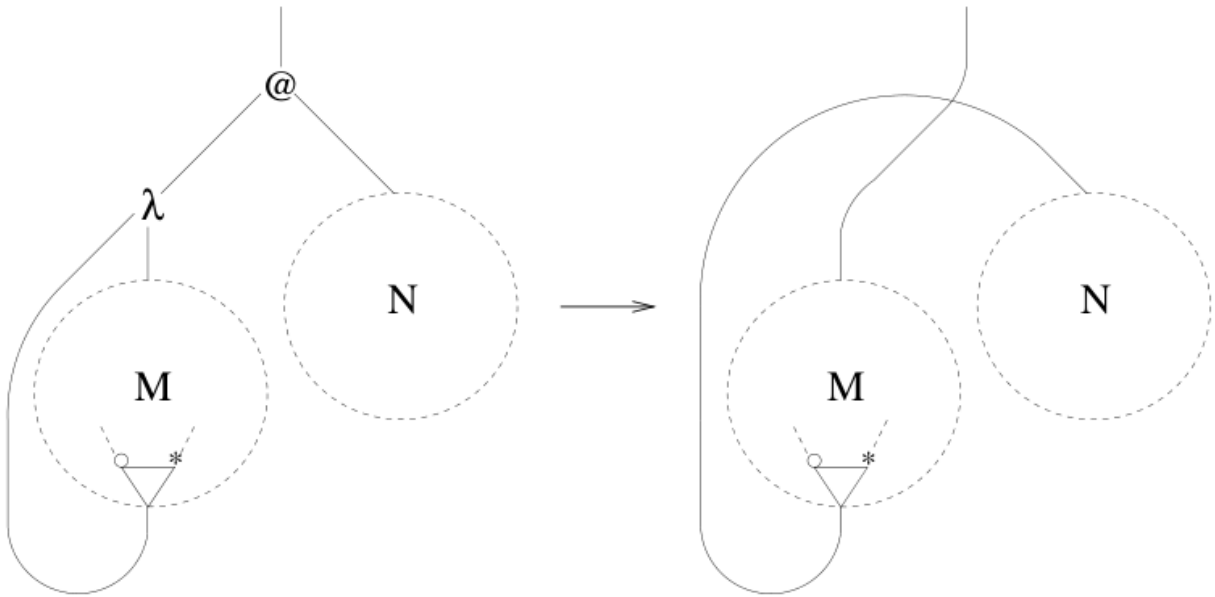


Fig. 2.9. β-reduction

In sharing graph reduction, substituting a variable x for a term N amounts to explicitly connect the variable to the term N. At the same time, the value returned by the application before firing the redex (the link above the application) becomes the instantiated body of the function (see Figure 2.9).

Figure: [1]

Dynamic system

β-reduction

The portions of graph representing M and N do not play any role in the sharing graph β-rule. In other words, β-reduction is expressed by the completely local graph rewriting rule of Figure 2.10.

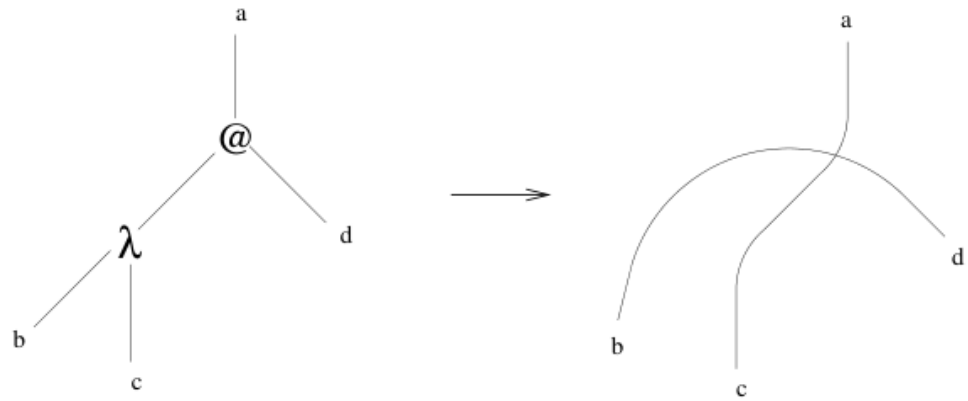


Fig. 2.10. β-rule

Figure: [1]

Dynamic system

β-reduction

```
pred beta_reduction [ap: Application]{
    -- Pre-conditions
    ap.func in Abstraction
    ap.status = Active
                                        ...
```

Dynamic system

β-reduction

```
...
-- Post-conditions
all a: Application | ((a.func = ap) =>
(a.func' = ap.func.body))

all a: Application | ((a.arg = ap) =>
(a.arg' = ap.func.body))

all a: Abstraction | ((a.body = ap) =>
(a.body' = ap.func.body))
...
```

Dynamic system

β-reduction

```
...
all a: (Application & subtree[ap.func.body]) |
    ((a.func = ap.func.param) =>
    (a.func' = ap.arg))

all a: (Application & subtree[ap.func.body]) |
    ((a.arg = ap.func.param) =>
    (a.arg' = ap.arg))

all a: (Abstraction & subtree[ap.func.body]) |
    ((a.body = ap.func.param) =>
    (a.body' = ap.arg))
...
```

Dynamic system

β-reduction

```
...
((ap.func not in subtree[ap.arg]) => (
    ap.func.param.status' = Inactive and
    ap.func.status' = Inactive   and
    ap.status' = Inactive))

-- Corner case
((ap.func in subtree[ap.arg]) =>
(ap.status' = Inactive))
...
```

Dynamic system

β-reduction

```
...
-- Frame conditions
((ap.func not in subtree[ap.arg])
=> (noStatusChange[Expression -
(ap + ap.func + ap.func.param)]))

((ap.func in subtree[ap.arg]) => (
    noStatusChange[Expression - ap]))
...
```

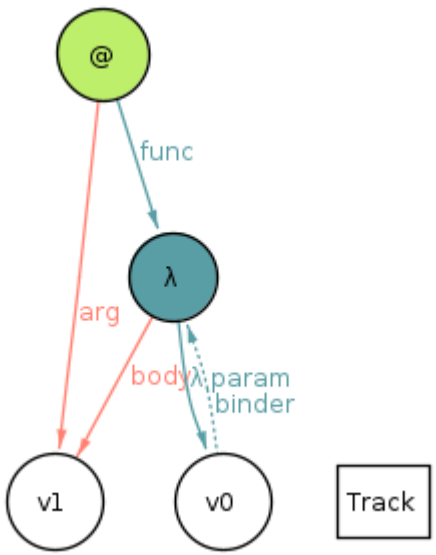
Dynamic system

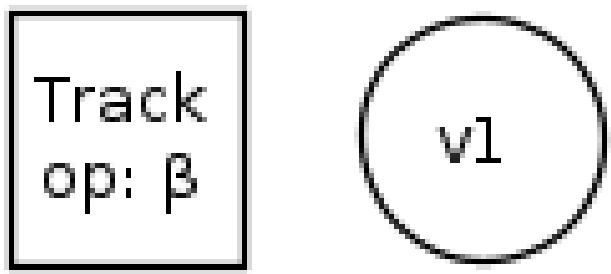
β-reduction

```
noParamChange[Abstraction]

noBodyChange[Abstraction
- ({a : Abstraction | a.body = ap}
+ {a: Abstraction & subtree [ap.func.body] |
a.body = ap.func.param})]

noFuncChange[Application
- ({a: Application | a.func = ap}
+ {a: Application & subtree[ap.func.body] |
a.func = ap.func.param })]

noArgChange[Application
- ({a: Application | a.arg = ap}
+ {a: Application & subtree[ap.func.body] |
a.arg = ap.func.param})]

noBinderChange[Variable]
```

Alloy's examples

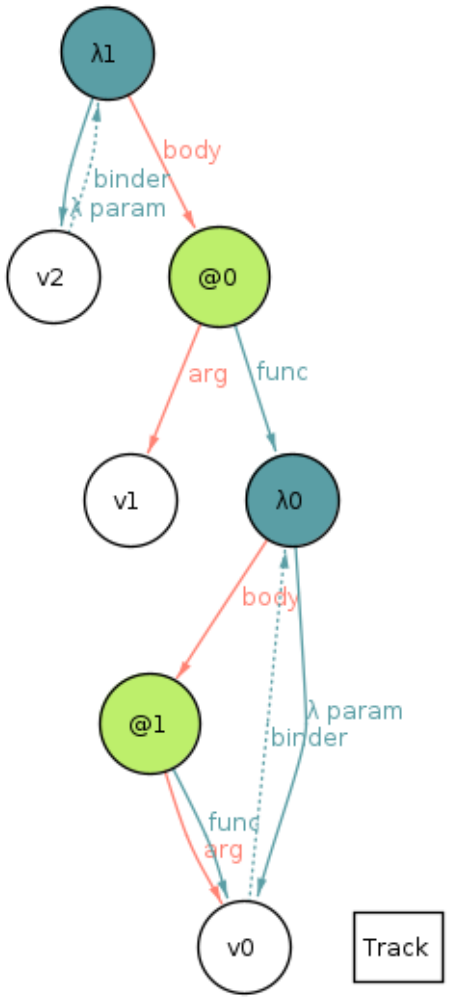
$$(\lambda x_0.x_1)x_1 \Rightarrow_\beta x_1$$

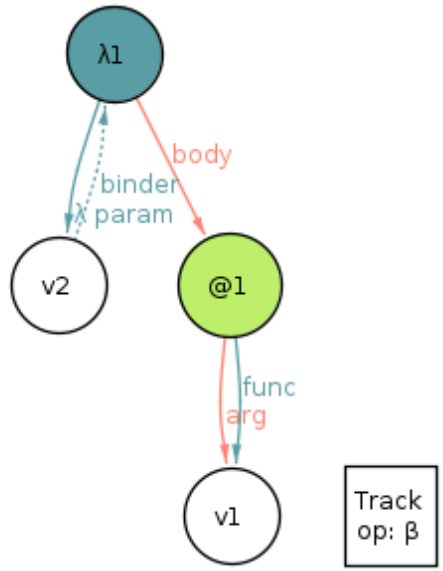




Alloy's examples

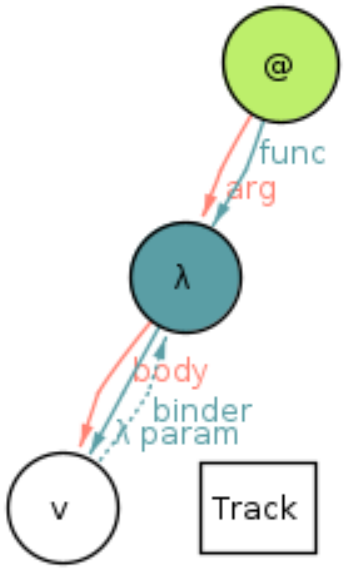
$$(\lambda x_2.((\lambda x_0.x_0 x_0)(x_1))) \Rightarrow_\beta (\lambda x_2.(x_1 x_1))$$

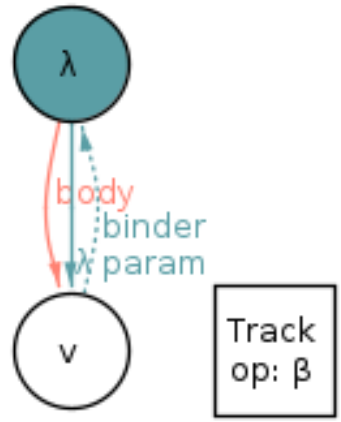




Alloy's examples

$$(\lambda x.x)(\lambda x.x) \Rightarrow_\beta \lambda x.x$$

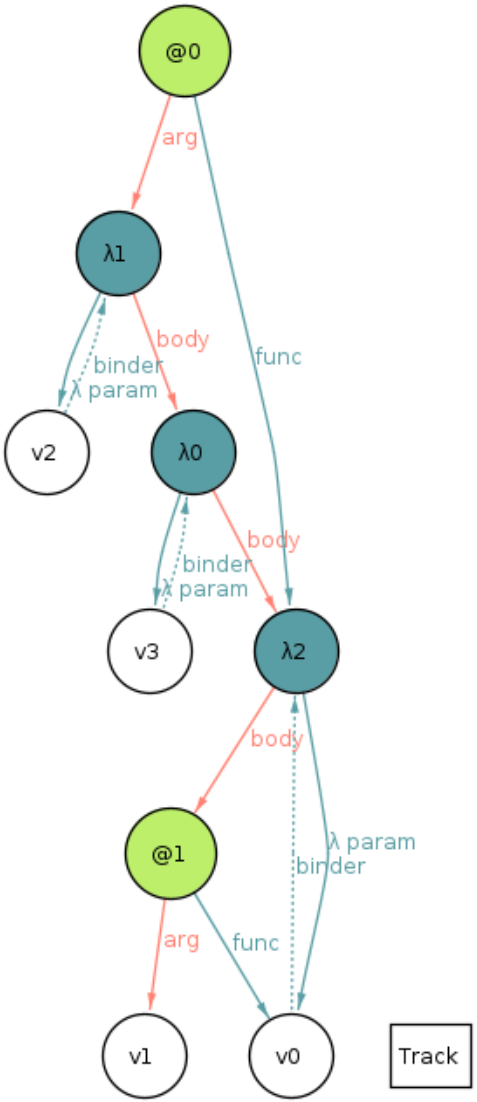


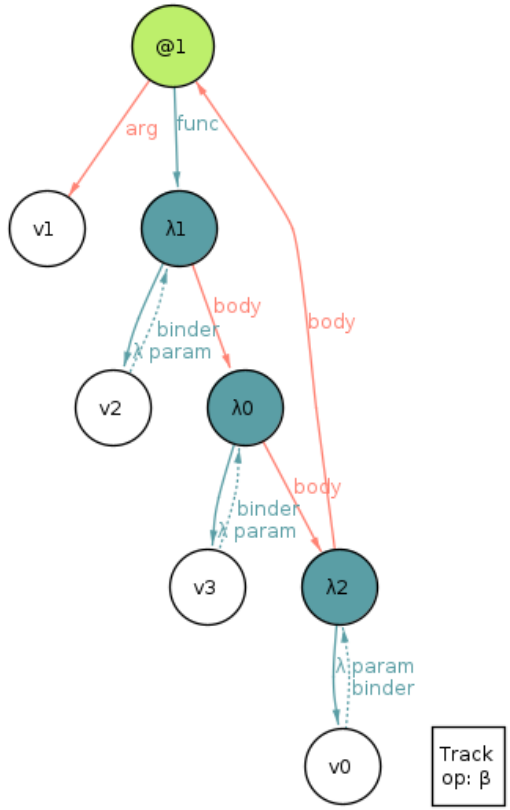


Alloy's examples

Corner case

`ap.func in subtree[ap.arg]` ou Ω ...





About the corner case

- Sometimes copying is necessary
- Handling reductions
- `var` signatures or *déjà*-structures
- Before it was turning off, now in the predicate it remains active

Assertions

```
pred p1 {
        eventually(always(Track.op != Beta))
}
```

Assertions

$$(\lambda x.x)T \equiv \mathbf{I}T \Rightarrow_\beta T$$

```
pred p2 {
-- Id N => N
(one ap : Application | (
 ap not in Expression.derivations and
 ap.func in Abstraction           and
 ap.func.body in Variable         and
 ap.func.param = ap.func.body     and
 ap.func != ap.arg                and
 beta_reduction[ap] )) =>
(one ap : Application | (
 beta_reduction[ap] and
 subtree[ap.arg] = active_expressions' ))}
```

Assertions

```
Executing "Check a2 for 8"
   Actual scopes: 2 Status, exactly 1 Active, exactly 1 Inact
   Solver=sat4j Steps=1..10 Bitwidth=4 MaxSeq=7 SkolemDepth=1
   1..10 steps. 1397333 vars. 18940 primary vars. 3500981 cla
   No counterexample found. Assertion may be valid. 20401ms.
```

Assertions

Next steps

- Handling reductions
- Fairness: Predicate for Ω
  `eventually(always(Track.op = Beta))`
- Liveness: Variable capture
- Structure of the reduction α

References

Andrea Asperti, Stefano Guerrini

The Optimal Implementation of Functional Programming Languages.

Cambridge University Press, 1998.

Beniamino Accattoli

Sharing a Perspective on the lambda Calculus.

ACM SIG- PLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software.

Cascais, Portugal, 2023.

H.P. Bandendregt

Functional Programming and Lambda Calculus.

Handbook of Theoretical Computer Science.

Elsevier, 1990.

Cap. 7.