

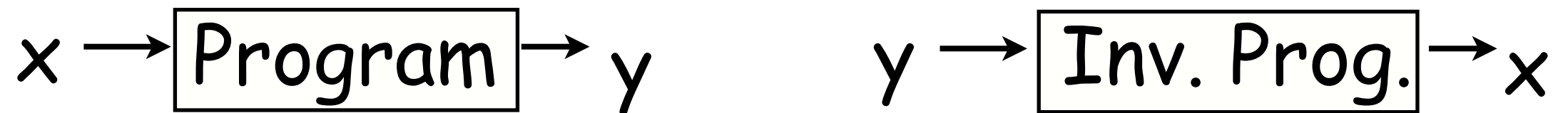
# A Grammar-based Approach to Invertible Programs

Kazutaka Matsuda (Univ. of Tokyo)  
Shin-Cheng Mu, Zhenjiang Hu, Masato Takeichi

Mar. 23th, ESOP 2010

# Background

- ▶ Writing a program that is inverse to some other program, such as ...
  - redoing for undoing
  - serialization for deserialization



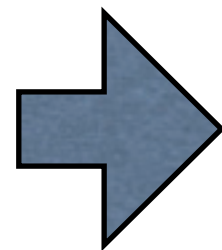
- ▶ ... is quite tedious and error-prone.

# Program Inversion

Given a program describing function  $f$ ,  
find a program describing  $f^{-1}$ .

e.g.

Program of  
 $\text{snoc}(x, b)$   
 $= x++[b]$



Inv. Prog. of  
 $\text{snoc}^{-1}(x++[b])$   
 $= (x, b)$

[Dijkstra: PC78, Gries: 81, Glück&Kawabe: FLOPS04, ...]

# This Talk

- ▶ **Grammar-based inversion**
  - uses grammars to approximate programs' evaluation.
    - production vs. evaluation
    - parsing vs. inverse computation
  - Advantages:
    - Clear classification of invertible programs.
    - Less heuristic.
    - Extensible.

# Target Programs

- ▶ First-Order Functional Programs with call-by-value semantics.

- e.g.: snoc

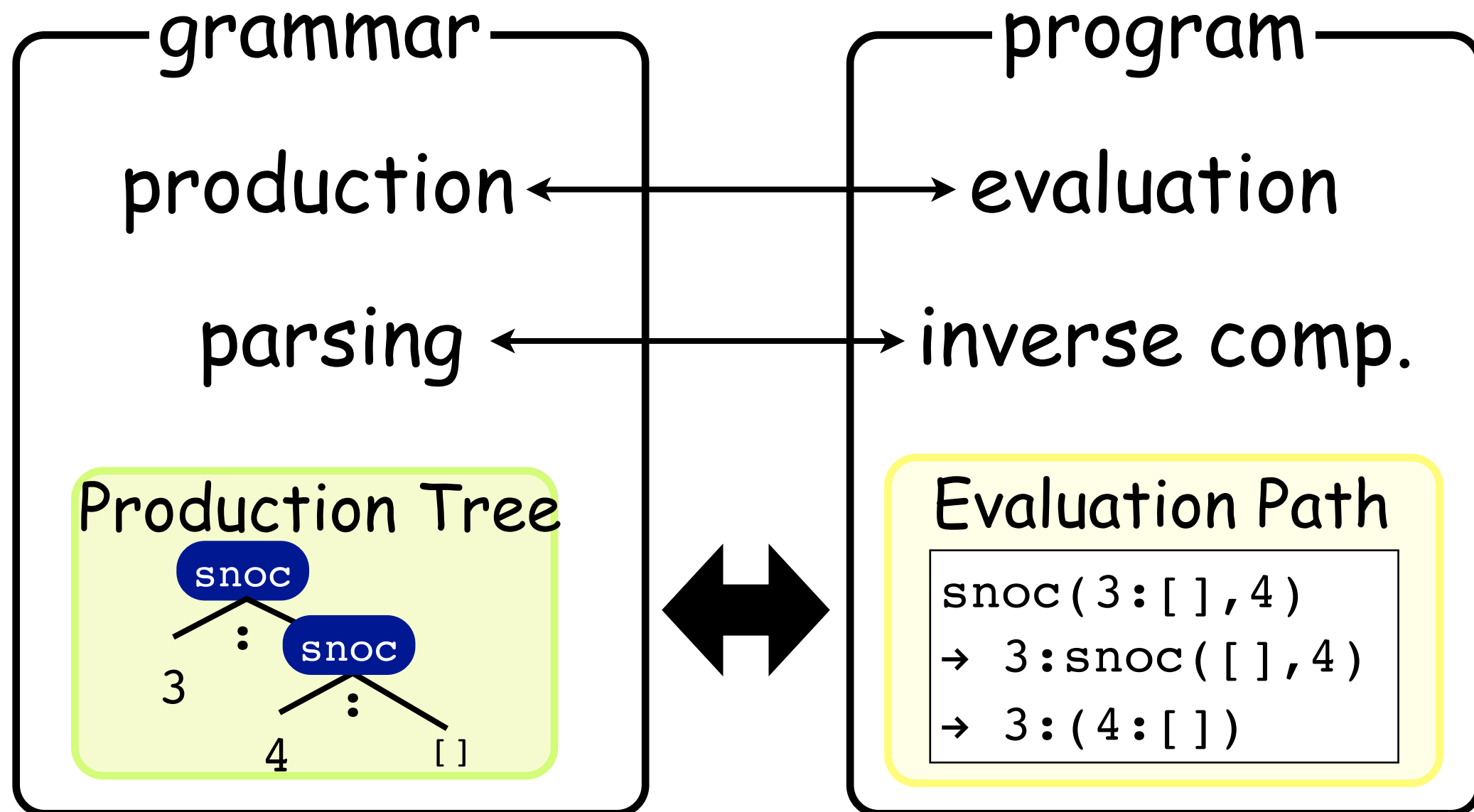
```
snoc( [], b ) = b : []  
snoc( a : x , b ) = a : snoc( x , b )
```

- example of evaluation

```
snoc( 3 : [] , 4 )  
→ 3 : snoc( [] , 4 )  
→ 3 : ( 4 : [] )
```

# Idea of Inversion

- ▶ Use grammar to approximate program's evaluation.



# Generation of Grammar

```
snoc([], b) = b:[]  
snoc(a:x, b) = a:snoc(x, b)
```

## Evaluation

```
snoc(3:[], 4)  
→ 3:snoc([], 4)  
→ 3:(4:[])
```

## ► Observation

- `snoc(...)` evaluates to `_:snoc(...)` or `_:[]`

# Generation of Grammar

```
snoc([], b) = b:[]  
snoc(a:x, b) = a:snoc(x, b)
```

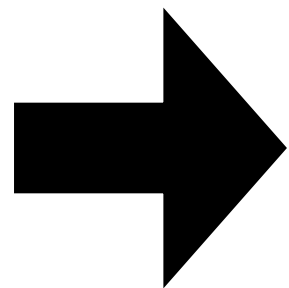
## Evaluation

```
snoc(3:[], 4)  
→ 3:snoc([], 4)  
→ 3:(4:[])
```

## ▶ Observation

- `snoc(...)` evaluates to `_:snoc(...)` or `_:[]`

## Grammar



```
snoc → ■:[]  
snoc → ■:snoc
```

Grammar represents  
func.-call traces.

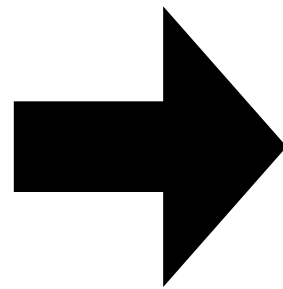


# Inverse Comp. by Parsing

```
snoc([], b) = b:[]  
snoc(a:x, b) = a:snoc(x, b)
```

## Evaluation

```
snoc(3:[], 4)  
→ 3:snoc([], 4)  
→ 3:(4:[])
```



## Grammar

```
snoc  
→ ■:[]  
snoc  
→ ■:snoc
```

$\text{snoc}^{-1}$

3:4:[]

# Inverse Comp. by Parsing

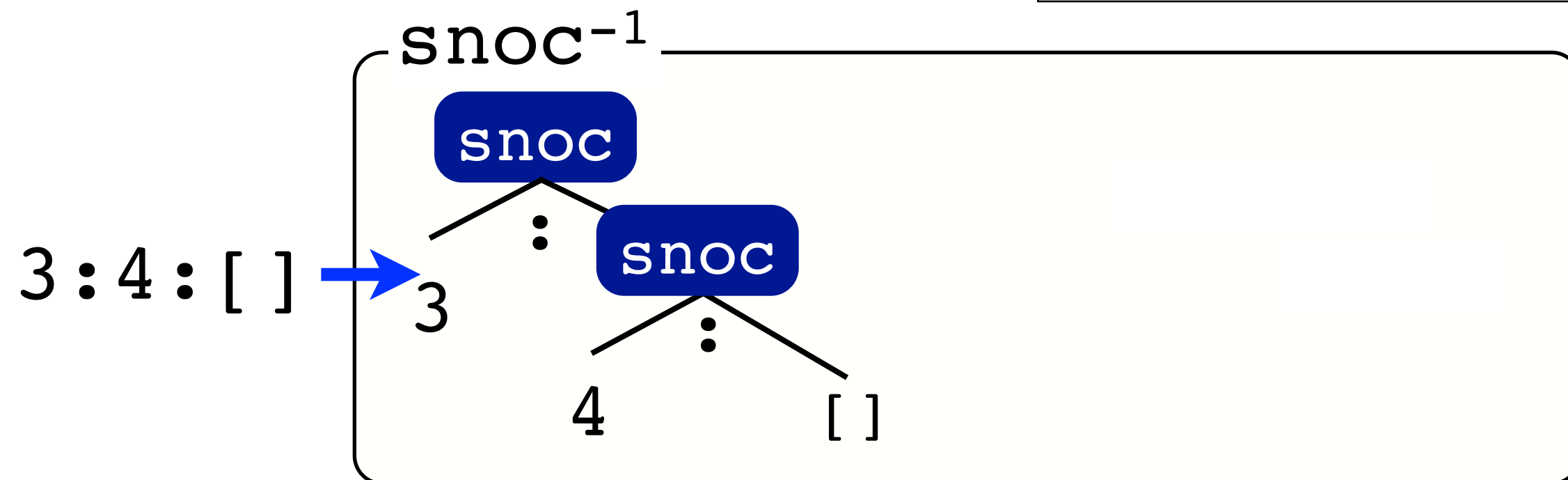
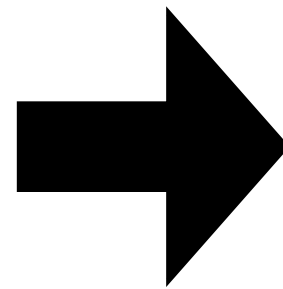
```
snoc([], b) = b:[]  
snoc(a:x, b) = a:snoc(x, b)
```

Evaluation

```
snoc(3:[], 4)  
→ 3:snoc([], 4)  
→ 3:(4:[])
```

Grammar

```
snoc  
→ ■:[]  
snoc  
→ ■:snoc
```



# Inverse Comp. by Parsing

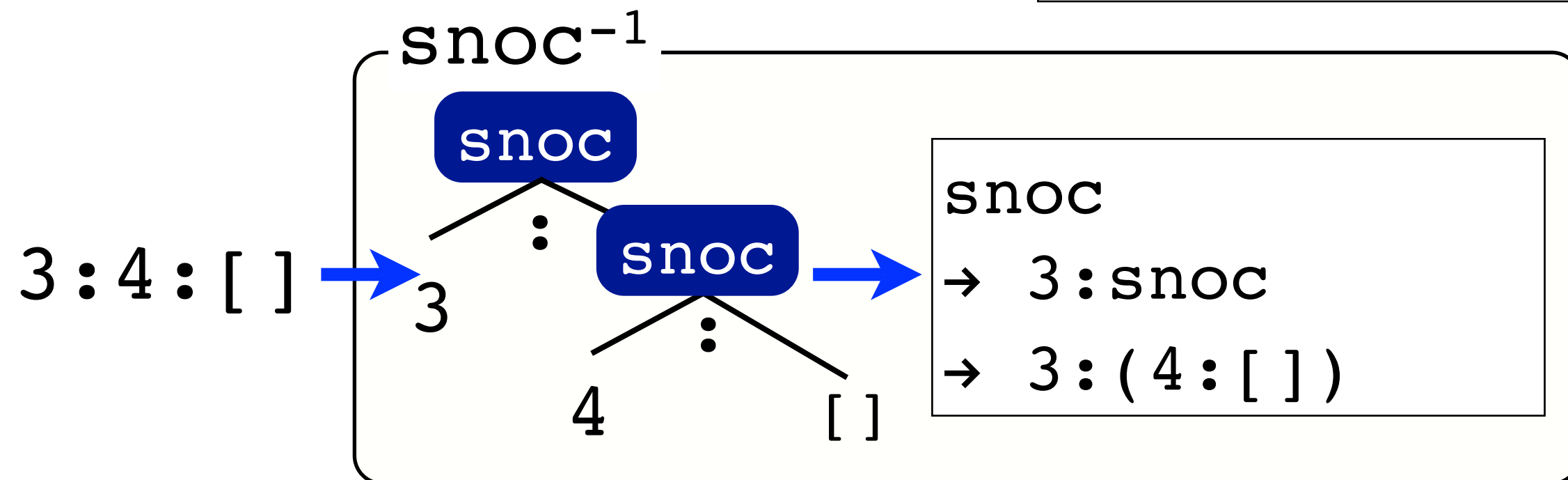
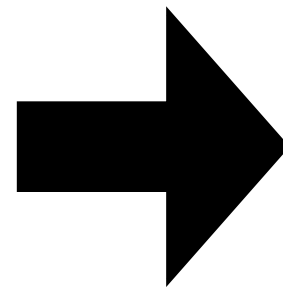
```
snoc([], b) = b:[]  
snoc(a:x, b) = a:snoc(x, b)
```

Evaluation

```
snoc(3:[], 4)  
→ 3:snoc([], 4)  
→ 3:(4:[])
```

Grammar

```
snoc  
→ ■:[]  
snoc  
→ ■:snoc
```



# Inverse Comp. by Parsing

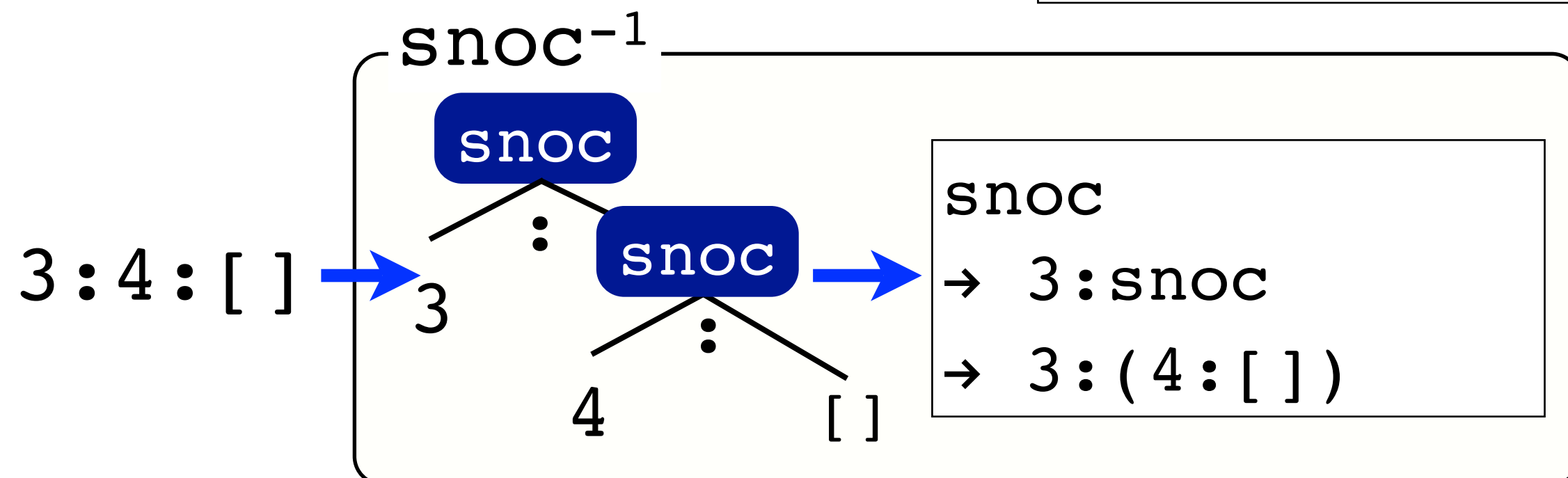
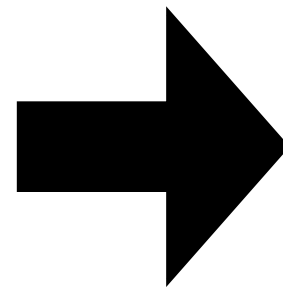
```
snoc([], b) = b:[]  
snoc(a:x, b) = a:snoc(x, b)
```

## Evaluation

```
snoc(3:[], 4)  
→ 3:snoc([], 4)  
→ 3:(4:[])
```

## Grammar

```
snoc([], b)  
→ b:[]  
snoc(a:x, b)  
→ a:snoc(x, b)
```



# Inverse Comp. by Parsing

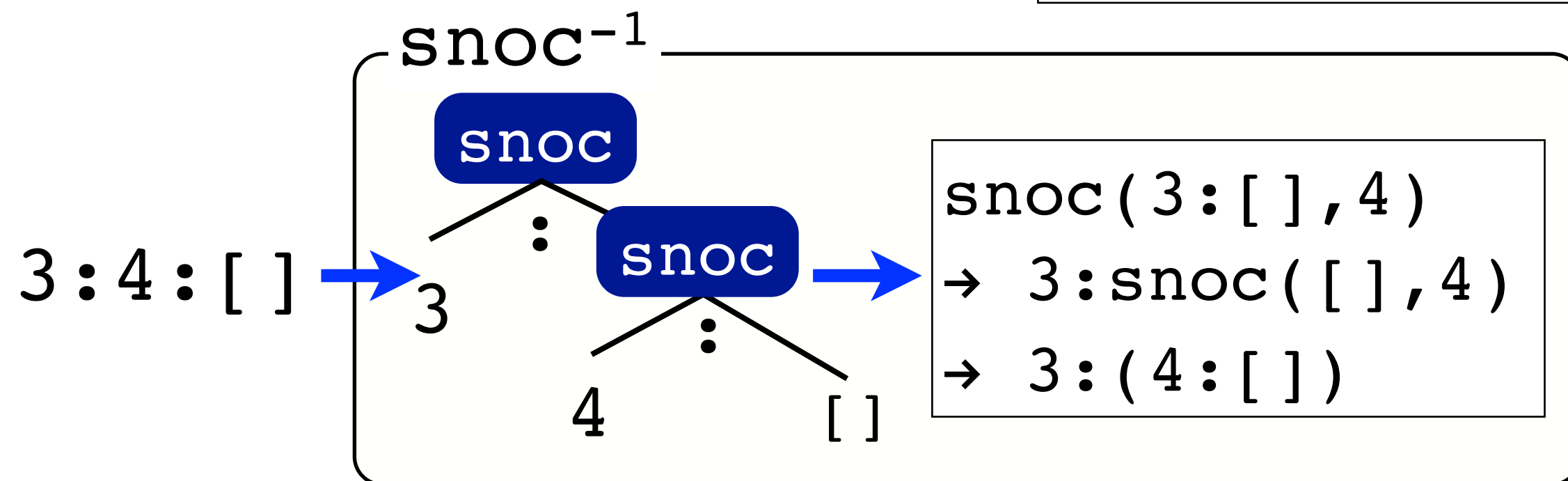
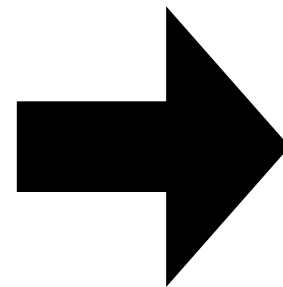
```
snoc([], b) = b:[]  
snoc(a:x, b) = a:snoc(x, b)
```

## Evaluation

```
snoc(3:[], 4)  
→ 3:snoc([], 4)  
→ 3:(4:[])
```

## Grammar

```
snoc([], b)  
→ b:[]  
snoc(a:x, b)  
→ a:snoc(x, b)
```



# Inverse Comp. by Parsing

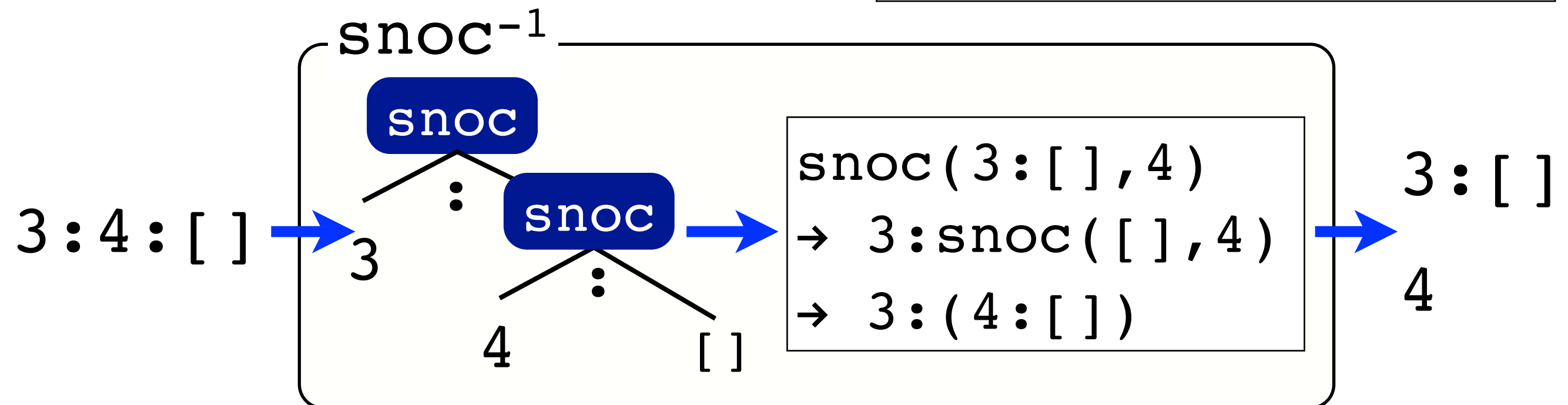
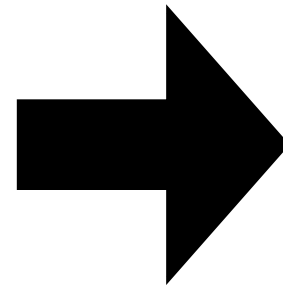
```
snoc([], b) = b:[]  
snoc(a:x, b) = a:snoc(x, b)
```

## Evaluation

```
snoc(3:[], 4)  
→ 3:snoc([], 4)  
→ 3:(4:[])
```

## Grammar

```
snoc([], b)  
→ b:[]  
snoc(a:x, b)  
→ a:snoc(x, b)
```



# Whole Inversion System

Input Program

`snoc(x, b) = ...`

Grammar

`snoc → ■ : snoc ...`

generate!

Invertible?

No

Fail

Yes

Output Inverse

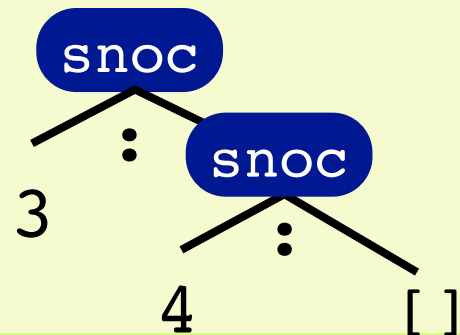
generate!

$snoc^{-1}(r) = \dots$

input of inverse

`3 : 4 : []`

Production Tree



reconstruct

Evaluation Path

`snoc(3 : [], 4)`  
`→ 3 : snoc([], 4)`  
`→ 3 : (4 : [])`

output of inverse

`(3 : [], 4)`

`snoc → ■ : snoc ...`

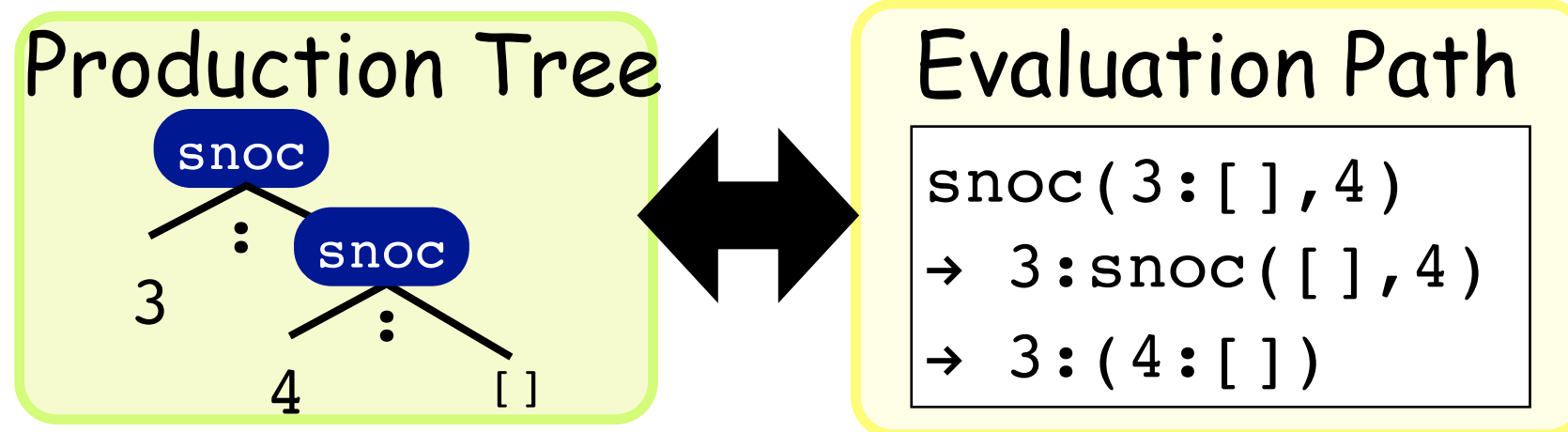
# Correctness

## Theorem

If a program is non-erasing and the derived grammar is **unambiguous**, then the derived inverse is actually inverse.

Proof.

**Bijection between production&evaluation**





# Efficiency

- ▶ The complexity of a derived inverse depends on the complexity of parsing.

Remark.

If a program is **linear** and **treeless**, the derived inverse **runs in  $O(n)$  time**, where  $n$  is the size of the input of the inverse.

- Treeless [Wadler: TCS90]
  - Every function-call has the form:  $f(x_1, \dots, x_n)$
  - e.g.: `snoc`

# Advantages

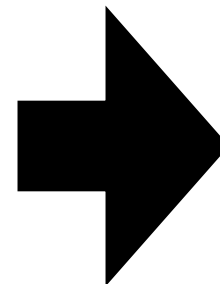
- ▶ Classification of invertible programs
- ▶ Less heuristic
- ▶ Extensible

# 1: Classification

- ▶ How hard it is to invert a program?  
← by grammar's hierarchy.

by Regular Tree Grammar

```
snoc( [], b ) = b : []  
snoc( a : x , b ) = a : snoc( x , b )
```

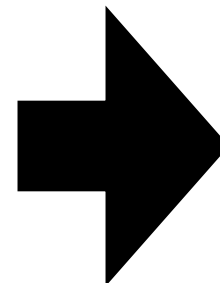


```
snoc → ■ : []  
snoc → ■ : snoc
```

by (IO) Context-Free Tree Grammar

[Engelfriet&Schmidt: J.Compt.Syst.Sci77]

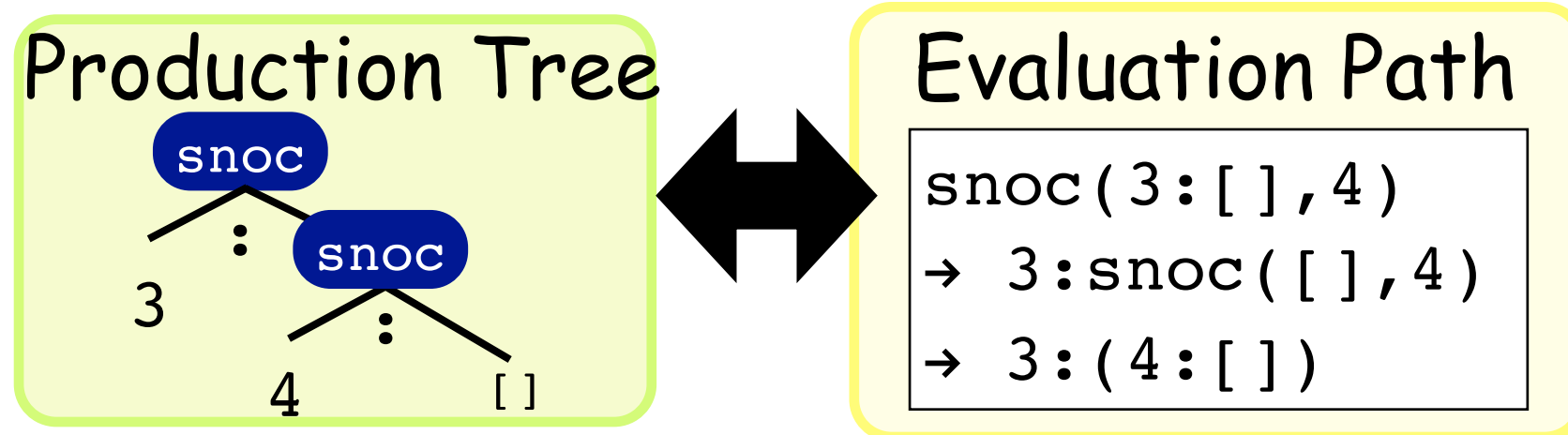
```
reverse( xs ) = rev( xs , [] )  
rev( [] , y ) = y  
rev( a : x , y ) = rev( x , a : y )
```



```
reverse → rev( [] )  
rev( y ) → y  
rev( y ) → rev( ■ : [] )
```

## 2: Less heuristic

- ▶ Grammar-based inversion is less heuristic.
  - based on the correspondence between evaluation/production.



# 3: Extensible

## ▶ Function compositions

```
snoc2(x,b,c) = snoc(snoc(x,b),c)
```

- Reparse the result of outer "snoc".

## ▶ Grammars

- Regular tree grammars
  - snoc, and treeless programs
- IO Context-free tree grammar
  - reverse, and

LMTTs [Engelfriet&Vogler: J.Compt.Syst.Sci85]

# Short Summary

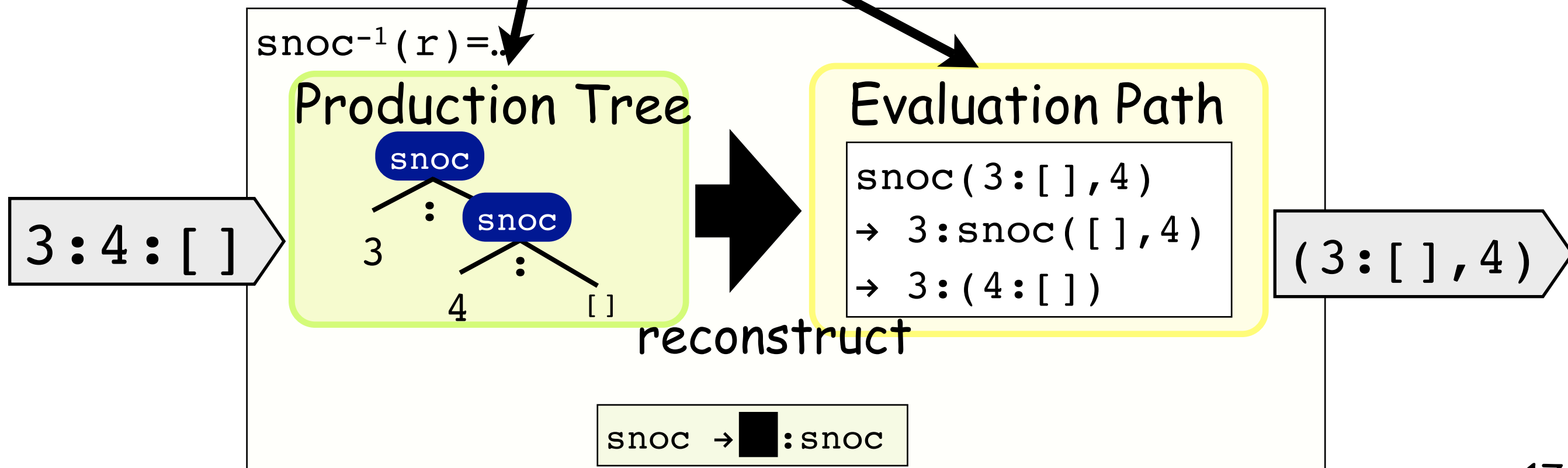
- ▶ **Grammar-based Inversion**
  - approximates program **evaluation** by grammar's **production**.
  - characterizes program's **invertibility** by grammar's **unambiguity**.
  - is based on **bidirectional** conversion of evaluation tree/production tree.

```
snoc( [],b) = b:[]  
snoc(a:y,b) = a:snoc(y,b)
```

```
snoc → ■:[]  
snoc → ■:snoc
```

# Is it efficient?

- ▶ A derived inverse seems to entail a lot of overhead due to large amount of intermediate data.



# Experiments

prototype impl.: <http://www.ipl.t.u-tokyo.ac.jp/~kztk/PaI/>

- ▶ Comparison to handwritten inverses.
  - to know overhead caused by "parsing".

Program	#input	handwritten	derived	speed ratio
snoc	8M	0.67s	0.95s	1.4
dbl	10M	0.11s	0.23s	2.1
zip	8M	0.28s	0.70s	2.5
runlength	9M	0.33s	0.76s	2.3

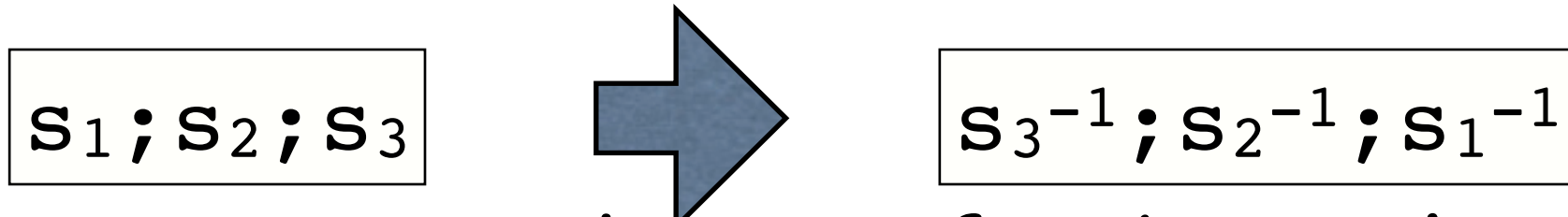
Acceptable overhead!



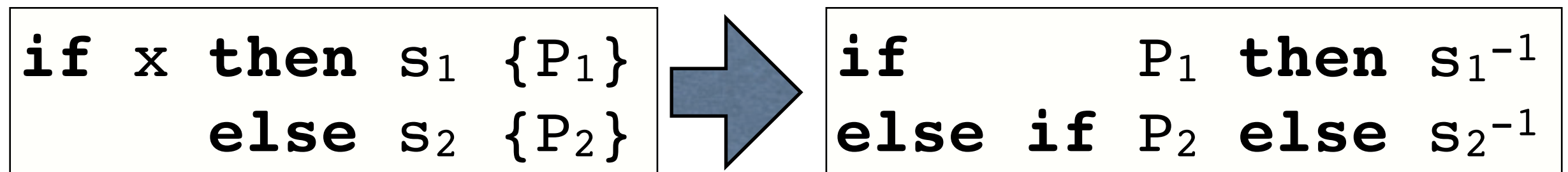
# Related Work

▶ Many of existing work

- revert execution order



- use post conditions for branches



▶ Our work is based on production/evaluation bijection.

# Closely Related Work

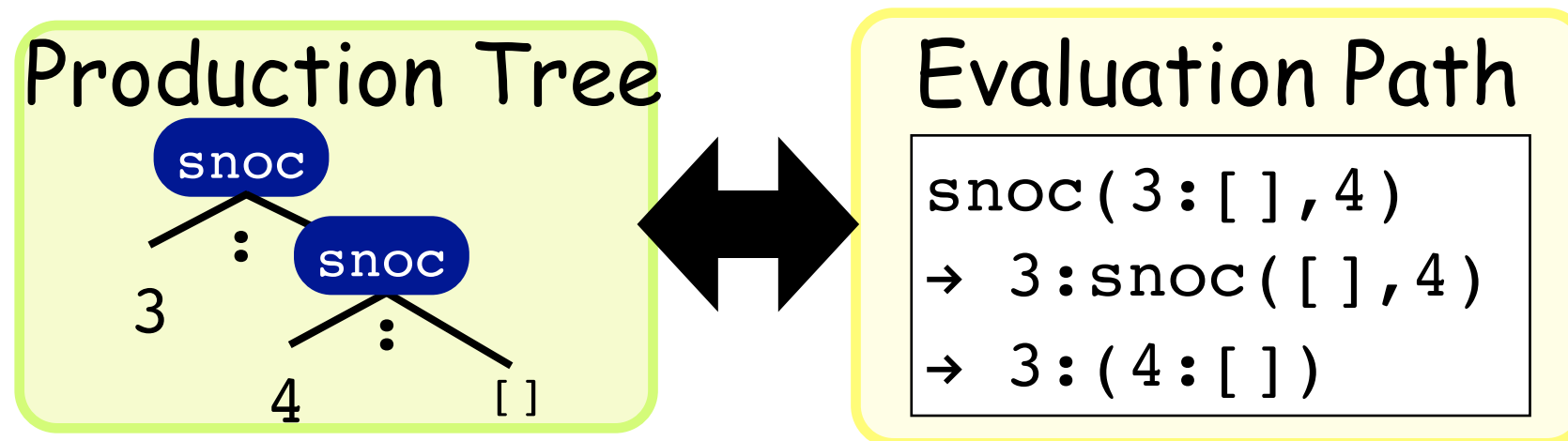
- ▶ [Glück&Kawabe: FLOPS04]
  - LR-parsing techniques to determine inverse programs
- ▶ [Yellin: ICSE88]
  - Program inversion of some class of AG
- ▶ Our work is a generalization and reformulation of these works.

# Conclusion

prototype impl.: <http://www.ipl.t.u-tokyo.ac.jp/~kztk/PaI/>

## ▶ Grammar-based Inversion

- ... is based on bijective mapping between **production** and **evaluation**.



- ... **classifies** programs by **unambiguous** grammars requires in inversion.
- **unambiguity** (+a) implies **invertibility**!

# Future Work

▶ More investigation on the idea

- $\exists x.f^{-1}(y) = x \Leftrightarrow y \in \text{Range}(f)$

▶ ... for right-inverses.



- Applications

- Compression & Decompression

- Testing:  $\forall x.P(x) \rightarrow Q(x)$  by using  $P^{-1}$

[Runciman et al.: Haskell08, Fischer: D.Thesis, ...]

- and so on.



# Programs

► First-Order Functional Programs with call-by-value semantics.

- e.g.: snoc

```
snoc( [], b ) = b : []  
snoc( a : x , b ) = a : snoc( x , b )
```

- example of evaluation

```
snoc( 3 : [], 4 )  
→ 3 : snoc( [], 4 )  
→ 3 : ( 4 : [] )
```

$$\left( \begin{array}{c} 3 : 4 : [] \\ \parallel \\ [ 3 , 4 ] \end{array} \right)$$

# Problem of Existing Methods

- ▶ It is unclear which programs are invertible

	dbl	snoc	reverse
[Glück&Kawebe: SIGPLAN Notices05]	OK	NG	NG
[Mogensen: GPCE 05]	OK	NG	NG
[Nishida&Sakai: ENTCS 09]	OK	OK	NG

[Glück&Kawebe: FLOPS 04]

```
reverse(x) = rev(x, [])  
rev(a:x, r) = rev(x, a:r)  
rev([], r) = r
```

```
dbl(z) = z  
dbl(S(x)) = S(S(dbl(x)))
```

```
snoc([], b) = b:[]  
snoc(a:x, b) = a:snoc(x, b)
```

# Questions we Answer

- ▶ Classification of invertible programs.
  - Which ones are essentially **easy** to invert, and which ones are **difficult**?
  - How **efficient** inverses can be derived for the programs?

```
reverse(x) = rev(x, [])  
rev(a:x, r) = rev(x, a:r)  
rev([], r) = r
```

```
dbl(z) = z  
dbl(S(x)) = S(S(dbl(x)))
```

```
snoc([], b) = b:[]  
snoc(a:x, b) = a:snoc(x, b)
```



# Classification Result

Every Invertible function

By Context-free Tree Grammar

`reverse`, ...

By Regular Tree Grammar

`snoc`, ...

(top-down deterministic)

`dbl`, ...



# Existing Methods

	dbl	snoc	reverse
[Glück&Kawebe: SIGPLAN Notices05]	OK	NG	NG
[Mogensen: GPCE 05]	OK	NG	NG
[Nishida&Sakai: ENTCS 09]	OK	OK	NG
[Glück&Kawebe: FLOPS 04]	OK	OK	OK

```
reverse(x) = rev(x, [])  
rev(a:x, r) = rev(x, a:r)  
rev([], r) = r
```

```
dbl(z) = z  
dbl(S(x)) = S(S(dbl(x)))
```

```
snoc([], b) = b:[]  
snoc(a:x, b) = a:snoc(x, b)
```

# Existing Methods

	dbl	snoc	reverse
[Glück&Kawebe: SIGPLAN Notices05]	OK	NG	NG
[Mogensen: GPCE 05]	OK	NG	NG
[Nishida&Sakai: ENTCS 09]	OK	OK	NG
[Glück&Kawebe: FLOPS 04]	OK	OK	OK

a bit Hard

```
reverse(x) = rev(x, [])
rev(a:x, r) = rev(x, a:r)
rev([], r) = r
```

Very Easy

```
dbl(z) = z
dbl(S(x)) = S(S(dbl(x)))
```

Easy

```
snoc([], b) = b:[]
snoc(a:x, b) = a:snoc(x, b)
```

# Issues of Program Inversion

- ▶ For efficient inverses, an inversion method often targets only a restricted class of programs.
- ▶ ... but **what class should we target?**

# Existing Method targets ...

	dbl	snoc	reverse
[Glück&Kawebe: SIGPLAN Notices05]	OK	NG	NG
[Mogensen: GPCE 05]	OK	NG	NG
[Nishida&Sakai: ENTCS 09]	OK	OK	NG
[Glück&Kawebe: FLOPS 04]	OK	OK	OK

```
reverse(x) = rev(x, [])  
rev(a:x, r) = rev(x, a:r)  
rev([], r) = r
```

```
dbl(z) = z  
dbl(S(x)) = S(S(dbl(x)))
```

```
snoc([], b) = b:[]  
snoc(a:x, b) = a:snoc(x, b)
```

# Existing Method targets ...

	dbl	snoc	reverse
[Glück&Kawebe: SIGPLAN Notices05]	OK	NG	NG
[Mogensen: GPCE 05]	OK	NG	NG
[Nishida&Sakai: ENTCS 09]	OK	OK	NG
[Glück&Kawebe: FLOPS 04]	OK	OK	OK

a bit Hard

```
reverse(x) = rev(x, [])
rev(a:x, r) = rev(x, a:r)
rev([], r) = r
```

Very Easy

```
dbl(z) = z
dbl(S(x)) = S(S(dbl(x)))
```

Easy

```
snoc([], b) = b:[]
snoc(a:x, b) = a:snoc(x, b)
```

# Existing Method targets ...

	dbl	snoc	reverse
[Glück&Kawebe: SIGPLAN Notices05]	OK	NG	NG
[Mogensen: GPCE 05]	OK	NG	NG
[Nishida&Sakai: ENTCS 09]	OK	OK	NG
[Glück&Kawebe: FLOPS 04]	OK	OK	OK

Is this result essential?

- intrinsic to the program?
- or inadequacy on the method?



# Question

- ▶ Classification of invertible programs.
  - Which ones are essentially **easy** to invert, and which ones are **difficult**?
  - How **efficient** inverses can be derived for the programs?

```
reverse(x) = rev(x, [])  
rev(a:x, r) = rev(x, a:r)  
rev( [], r) = r
```

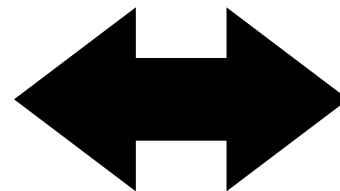
```
dbl(z) = z  
dbl(S(x)) = S(S(dbl(x)))
```

```
snoc( [], b) = b:[]  
snoc(a:x, b) = a:snoc(x, b)
```

# Our Approach

- ▶ **Grammar-based Inversion**
  - ... approximates program **evaluation** by grammar's **production**.
  - ... characterizes program's **invertibility** by grammar's **unambiguity**.
  - ... is based on **bidirectional** conversion of evaluation tree/production tree.

```
snoc(a:y,b) = a:snoc(y,b)
snoc( [],b) = b:[]
```



```
snoc → ■ : [ ]
snoc → ■ : snoc
```

# Our Answer to the Question

- ▶ Classification of invertible programs.

- Which ones are essentially **easy** to invert, and which ones are **difficult**?

← **By how complex grammar is used**

- How **efficient** inverses can be derived for the programs?

← **By the computational complexity of parsing with the grammar**

# Classification Result

Every Invertible function

By Context-free Tree Grammar

`reverse`, ...

By Regular Tree Grammar

`snoc`, ...

(top-down deterministic)

`dbl`, ...

Idea of

**Grammar-Based Inversion**

