Optimality for Dynamic Patterns

Thibaut Balabonski

Laboratoire PPS
CNRS and Université Paris Diderot

PPDP 2010: July 28

Computing with data structures

```
type 'a tree =
  | Data of 'a
  | Node of 'a tree * 'a tree
```

```
let rec upd f t = match t with
  | Data a   -> Data (f a)
  | Node l r -> Node (upd f l) (upd f r)
```

Pattern matching:

- tests the shape.
- extracts substructures.

New polymorphisms

- Pattern polymorphism: parameters in patterns.
- Path polymorphism: shape irrelevance.

```
let rec upd c f t = match t with
  | c d -> c (f d)
  | x y -> (upd c f x) (upd c f y)
  | a   -> a
```

Instantiation of the parameter c

- Constructors like `Data`.
- **Functions building patterns**.

New polymorphisms

- Pattern polymorphism: parameters in patterns.
- Path polymorphism: shape irrelevance.

```
let rec upd c f t = match t with
  | Data d -> Data (f d)
  | x y -> (upd c f x) (upd c f y)
  | a   -> a
```

Instantiation of the parameter c

- Constructors like Data.
- **Functions building patterns.**

New polymorphisms

- Pattern polymorphism: parameters in patterns.
- Path polymorphism: shape irrelevance.

```
let rec upd c f t = match t with
  | (fun x -> Node (Data x) (Data _)) d ->
  | x y -> (upd c f x) (upd c f y)
  | a   -> a
```

Instantiation of the parameter c

- Constructors like Data.
- **Functions building patterns.**

New polymorphisms

- Pattern polymorphism: parameters in patterns.
- Path polymorphism: shape irrelevance.

```
let rec upd c f t = match t with
  | Node (Data d) (Data _) -> Node ...
  | x y -> (upd c f x) (upd c f y)
  | a   -> a
```

Instantiation of the parameter c

- Constructors like Data.
- **Functions building patterns.**

New polymorphisms

- Pattern polymorphism: parameters in patterns.
- Path polymorphism: shape irrelevance.

```
let rec upd c f t = mat
  | c d -> c (f d)
  | x y -> (upd        upd c f y)
  | a   -> a
```

Dynamic patterns

Inst of the parameter c

- C ructors like Data.
- **Functions building patterns**.

Collateral damage

Usual compile-time analyzes

```
| Node  l          (Data d)   ->
| Node (Data d)    (Node _ _) ->
| Node (Node l r) (Node _ _) ->
```

Collateral damage

Usual compile-time analyzes

```
| Node  l          (Data d)   ->
| Node (Data d)    (Node _ _) ->
| Node (Node l r)  (Node _ _) ->
```

Collateral damage

Usual compile-time analyzes

```
| Node  l          (Data d)   ->
| Node (Data d)    (Node _ _) ->
| Node (Node l r)  (Node _ _) ->
```

This requires static patterns!

Goal

Run-time sharing mechanisms
for dynamic patterns.

Roadmap

1. Is optimal sharing a reasonable solution?
2. Families of terms: learning from history
3. (Some syntax)
4. Recording the history
5. From history to memory

Is optimal sharing a reasonable solution?

There is some mismatch between theoretical optimality and practical efficiency.

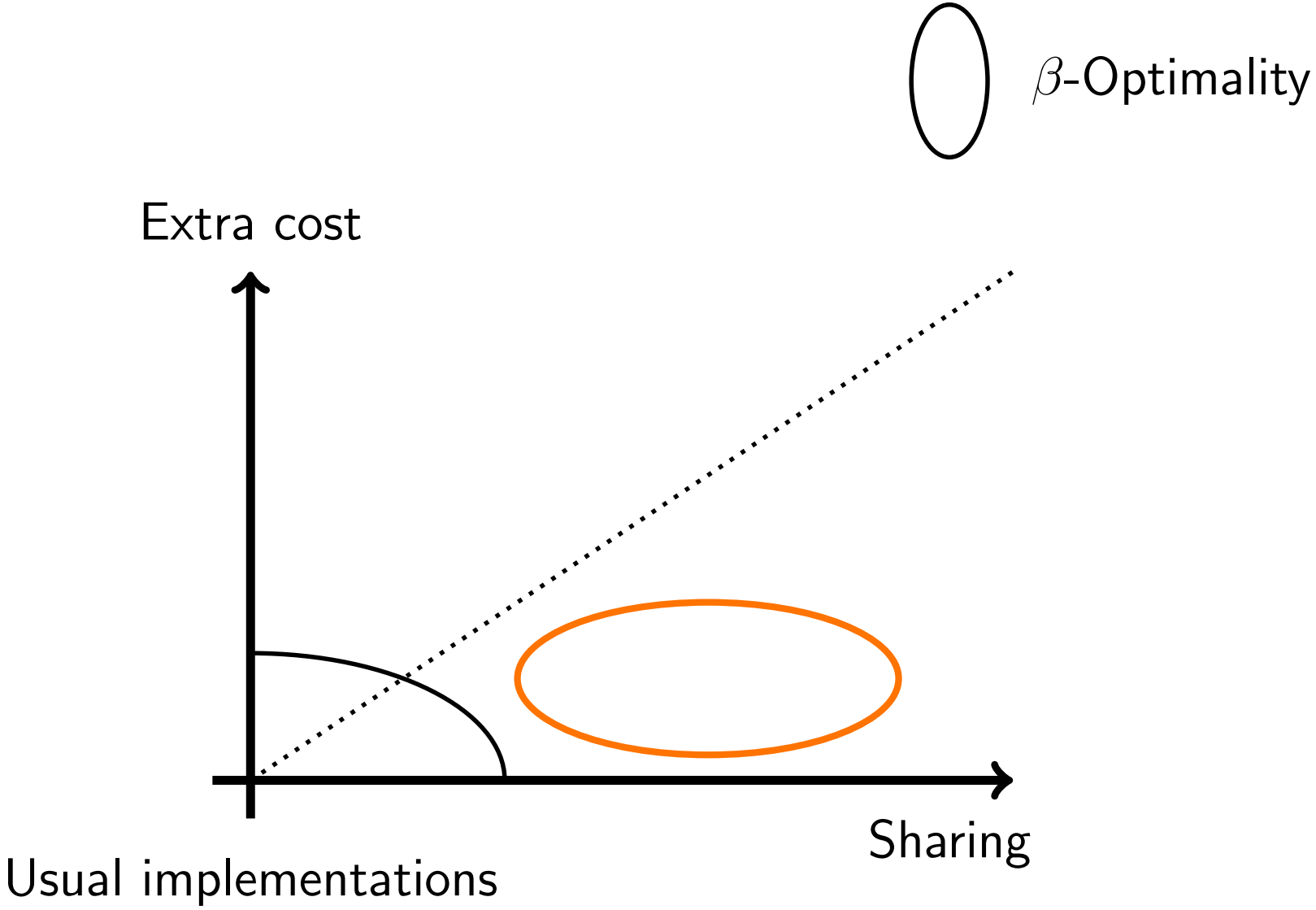
Families of terms: learning from history

(Some syntax)

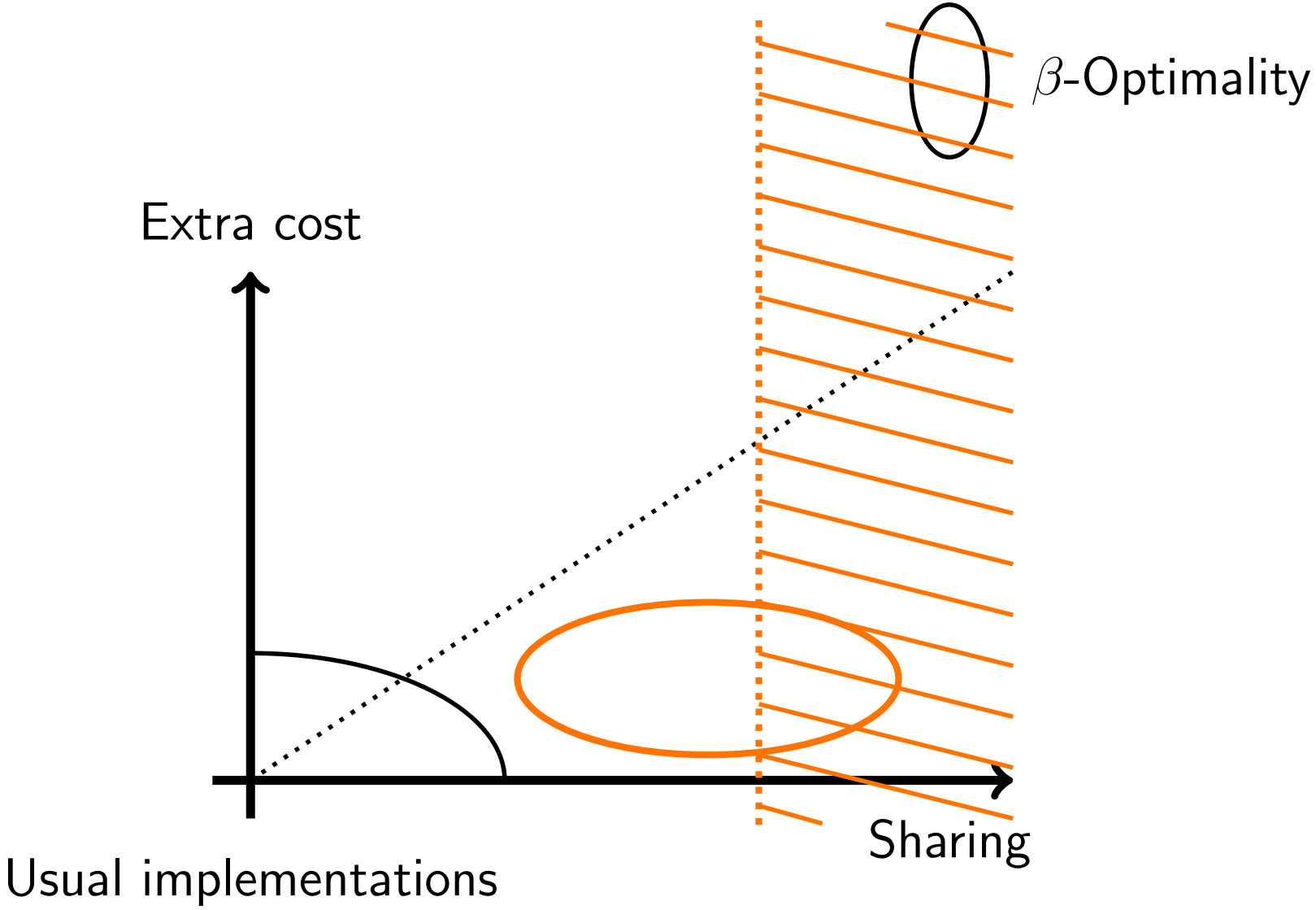
Recording the history

From history to memory

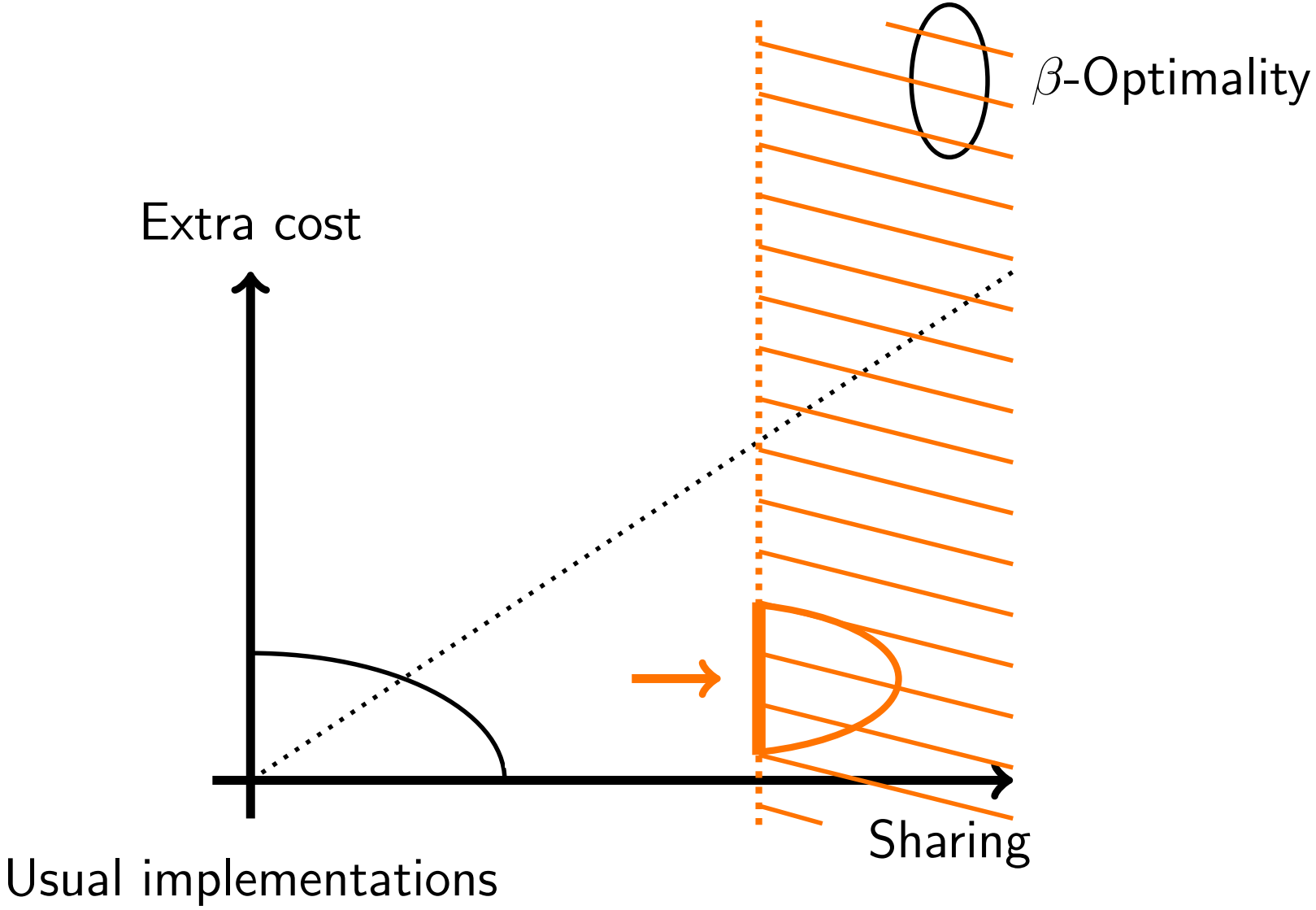
The trap of optimality



The trap of optimality



The trap of optimality



Weak reduction

Usual implementations

- Passing arguments to functions.
- No partial evaluation.

$\Rightarrow$ No evaluation under a binder (λ).

Confluent weak reduction

No reduction between
a bounded variable and its binder.

Families of terms: learning from history

Terms with same history should not be considered different.

(Some syntax)

Recording the history

From history to memory

Optimality and history

Optimality commandment

At each step you shall reduce a whole family.

- **Family**: all redexes sharing a common history.
- **History**: all events that contributed to the creation of a redex.
- **Redex**: REDucible EXpression.

Screening the history

Which reduction steps are relevant?

$\longrightarrow \quad \longrightarrow \quad \longrightarrow \quad \longrightarrow \quad \longrightarrow \quad \longrightarrow \quad \longrightarrow \quad \longrightarrow \quad \longrightarrow \quad \longrightarrow$

Back to basics

- analysis of direct contribution.
- how can redexes be created?

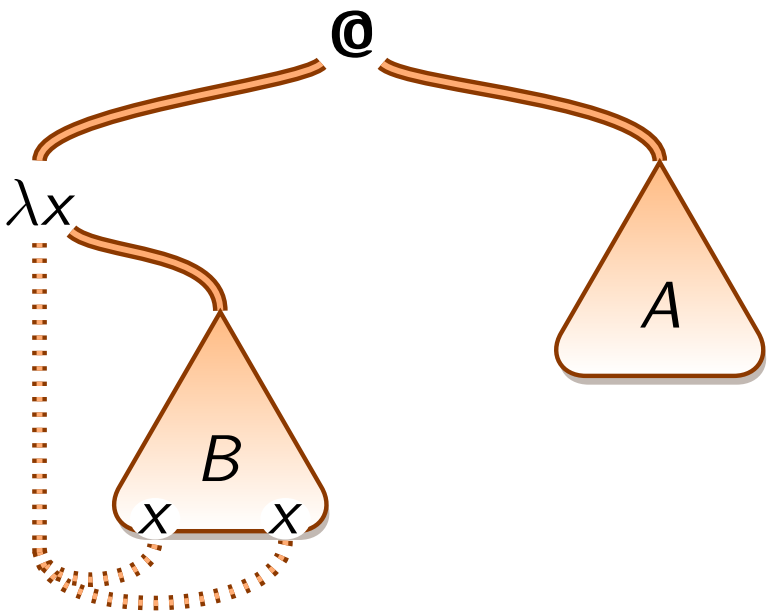
And later

- record contributions in labels.

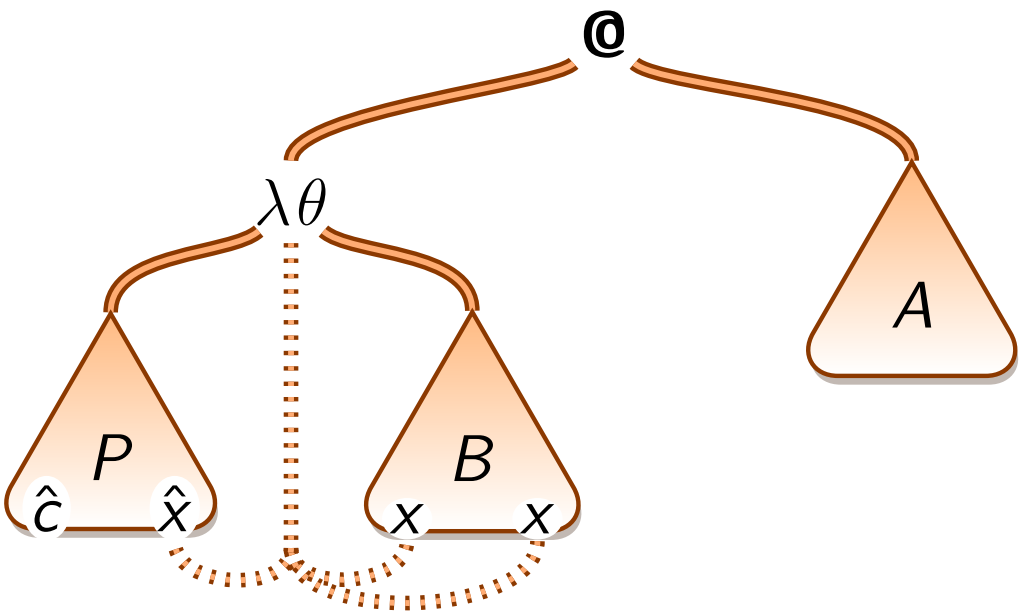
(Some syntax)

Digression: the Weak Pure Pattern Calculus.
Simple syntax, complex reduction.

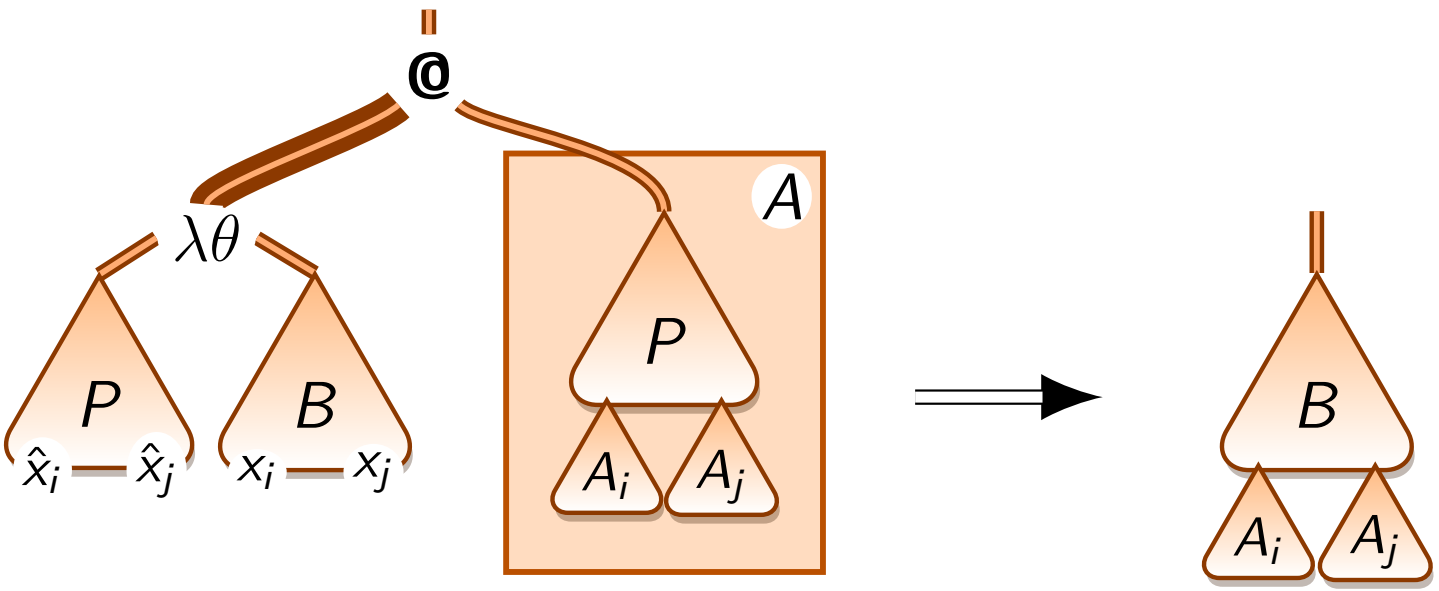
Syntactic trees



Syntactic trees



Pattern matching: a complex operation



Failure

If A and P incompatible, reduction to $\perp$.

Partial definition

Matching undefined for improper terms.

Pattern matching: a meta-operation

$$
\begin{array}{rcll}
\{\!\{a/_\theta\, \hat{x}\}\!\} & = & \{x \leftarrow a\} & x \in \theta \\
\{\!\{\hat{x}/_\theta\, \hat{x}\}\!\} & = & \{\} & x \notin \theta \\
\{\!\{a_1 a_2/_\theta\, p_1 p_2\}\!\} & = & \{\!\{a_1/_\theta\, p_1\}\!\} \uplus \{\!\{a_2/_\theta\, p_2\}\!\} & a_1 a_2,\ p_1 p_2 \text{ matchable} \\
\{\!\{a/_\theta\, p\}\!\} & = & \bot & a,\ p \text{ matchable}
\end{array}
$$

Undefined otherwise!

Matchable forms

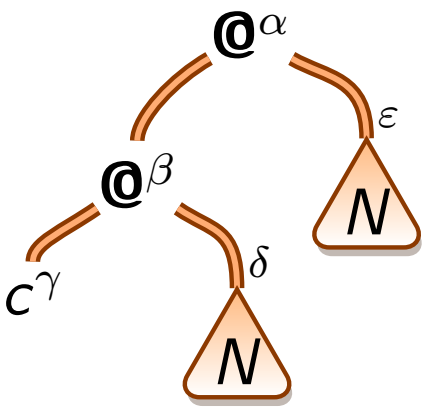
- $\hat{x}\ t_1\ \ldots\ t_n$
- $[\theta]p \rightarrow b$

Recording the history

Labels to record contributions
and define redex families.

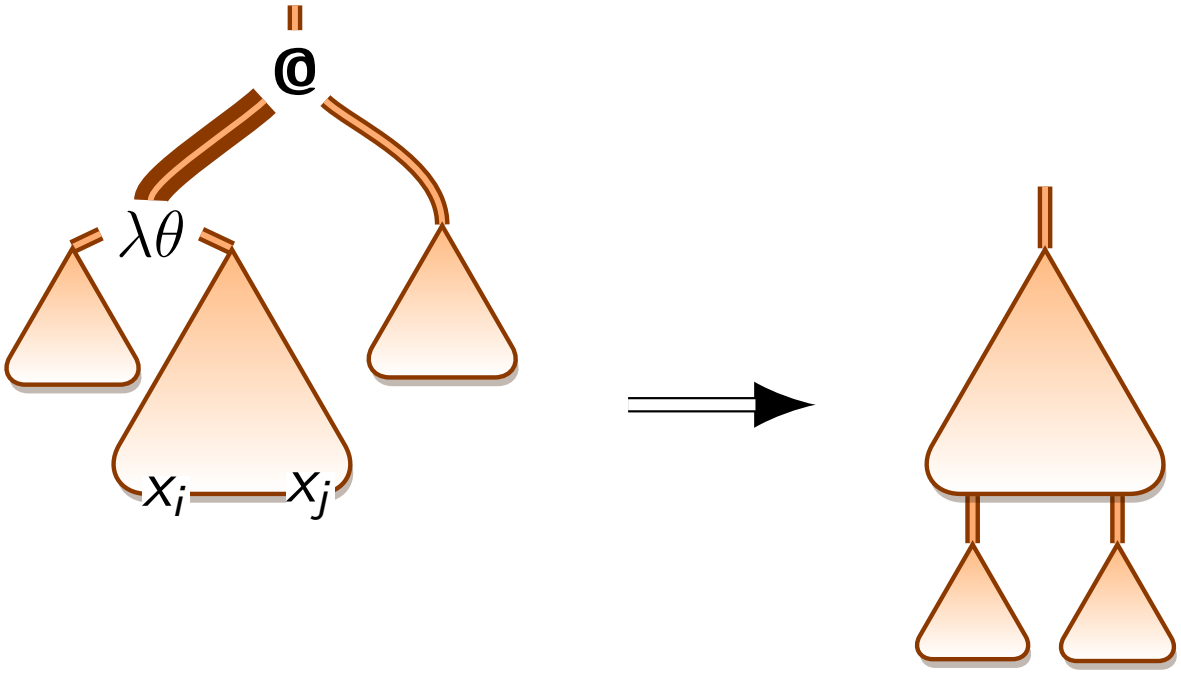
History and contributions materialized

Contributions can be recorded by labels.

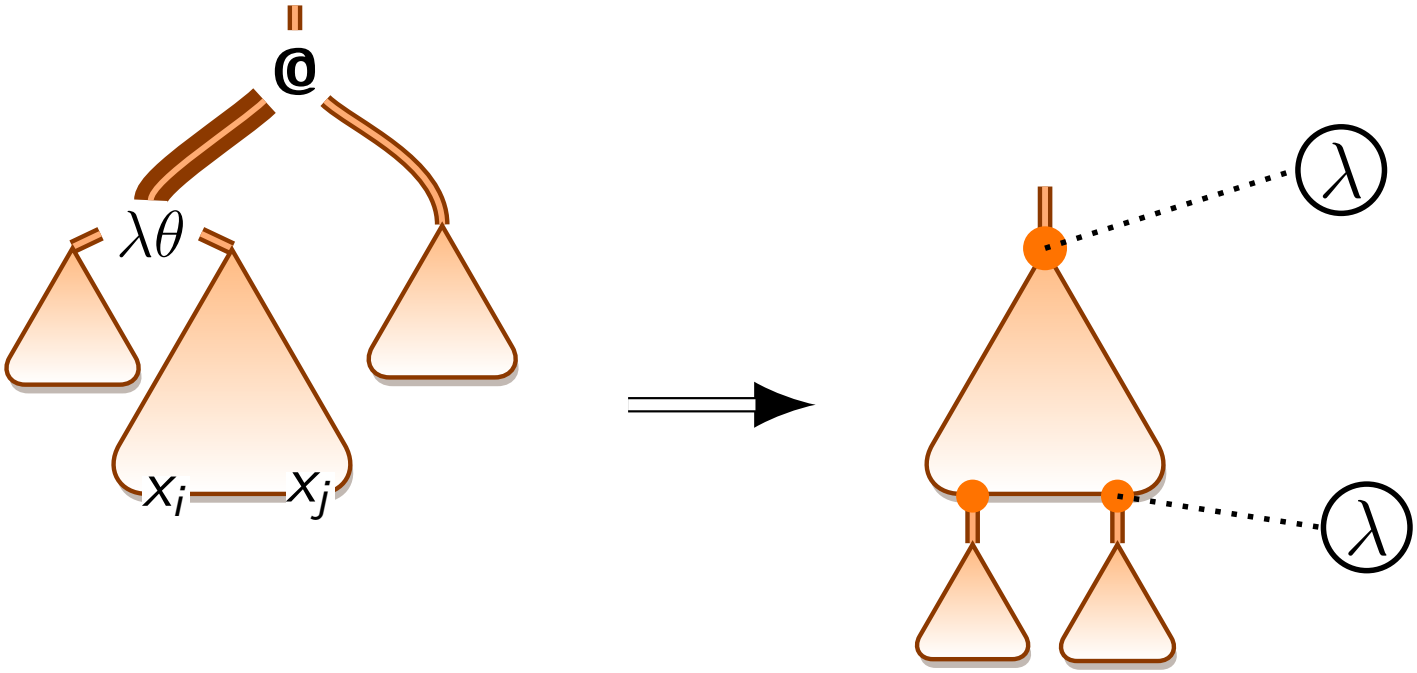


- **Family**: all redexes sharing a common label.

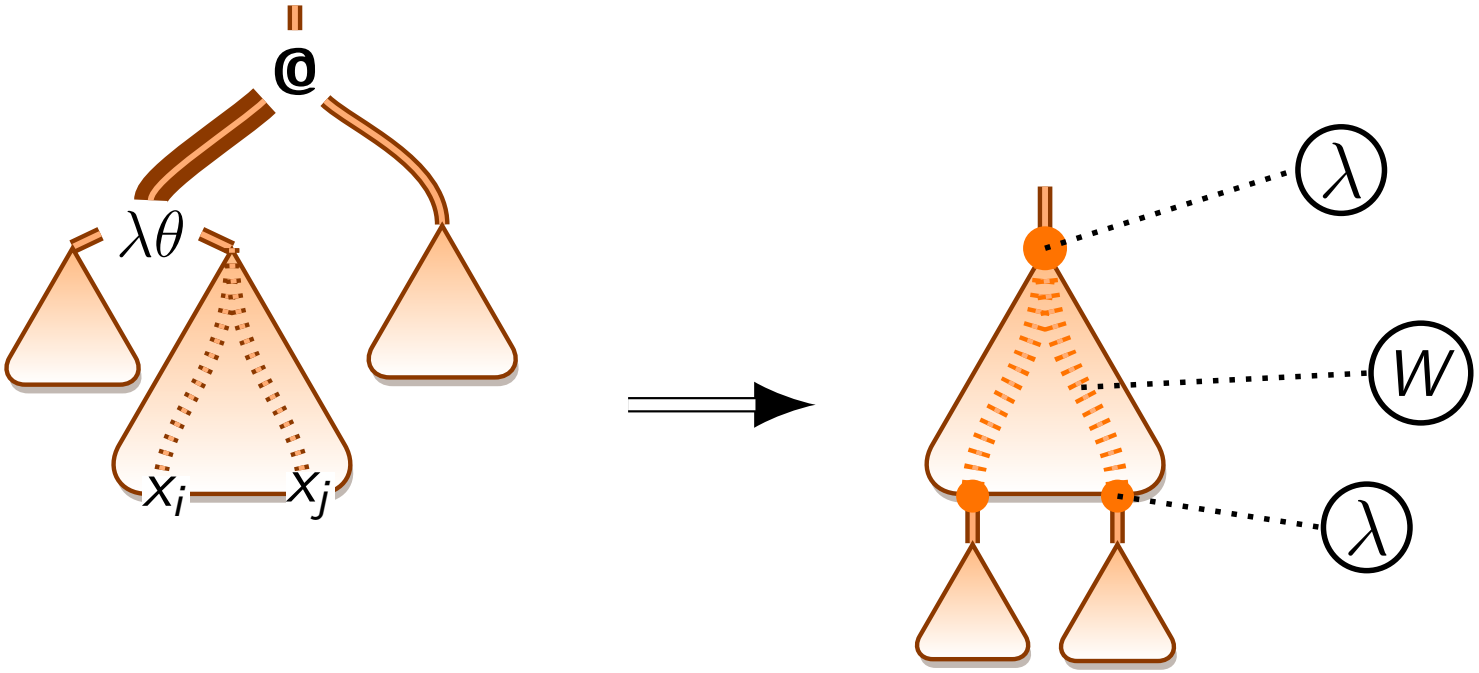
Forward: The consequences of my deeds



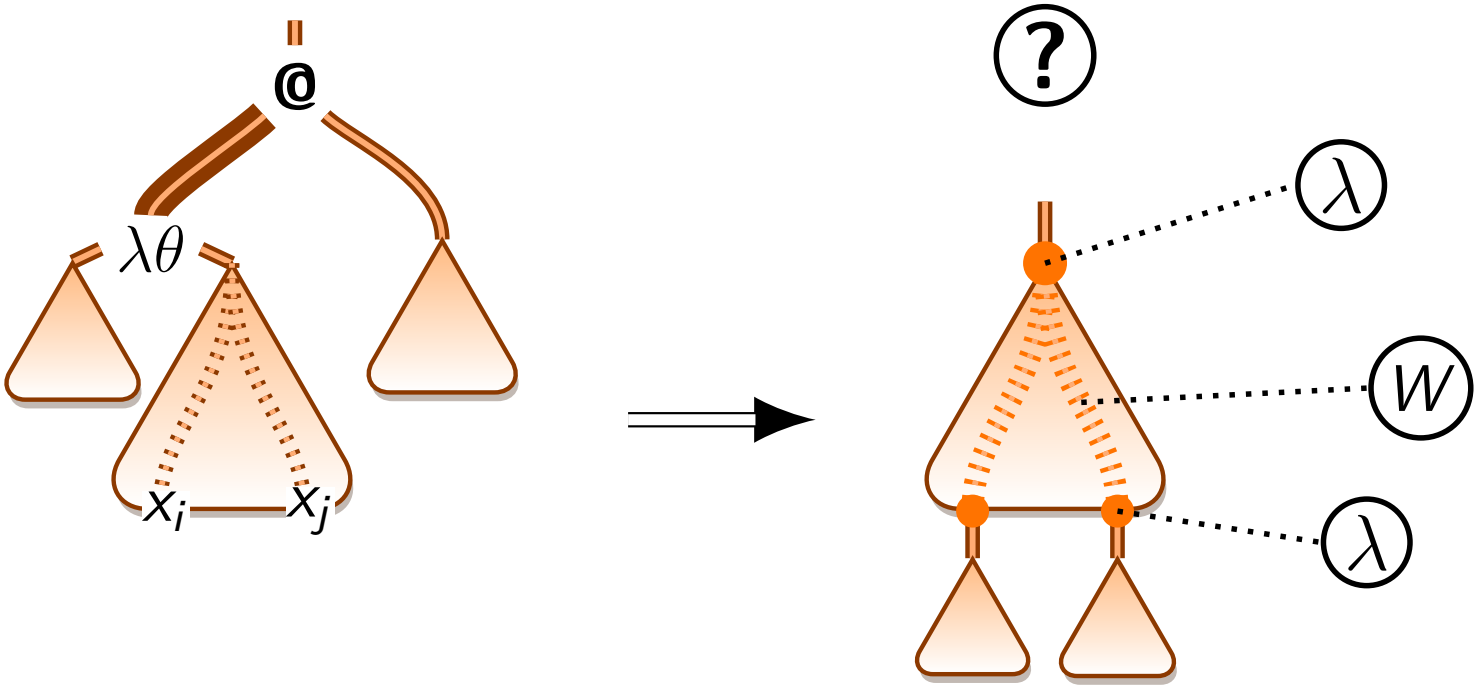
Forward: The consequences of my deeds



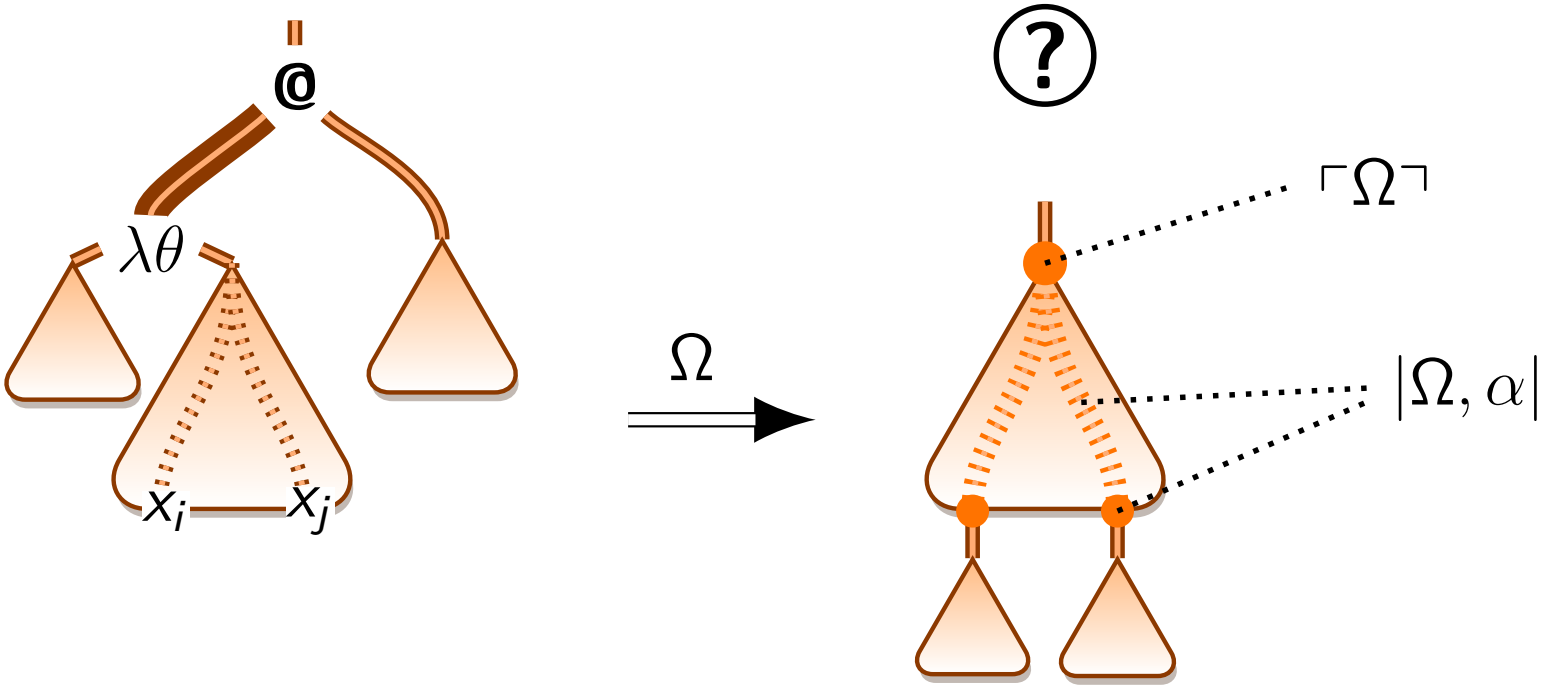
Forward: The consequences of my deeds



Forward: The consequences of my deeds

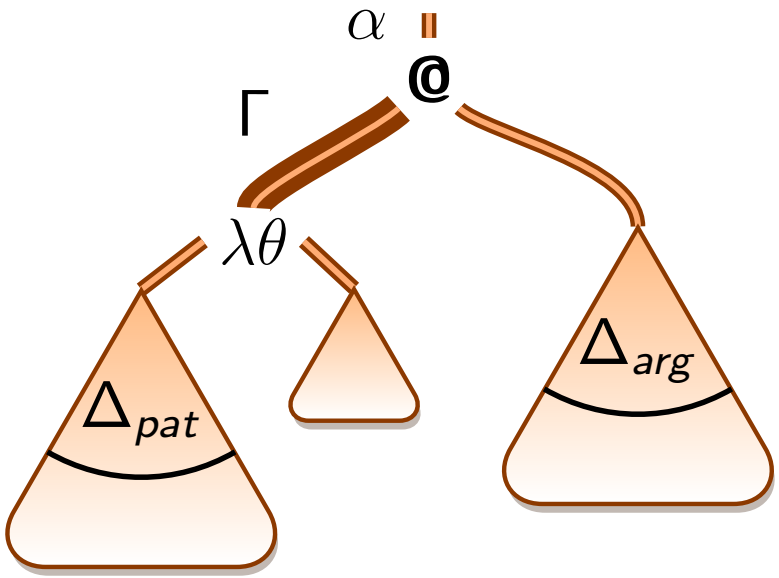


Forward: The consequences of my deeds



initial labels	$\mathbb{a}, \mathbb{b}, \mathbb{g} \ldots$		
atomic labels	α	$::=$	$\mathbb{a} \mid \ulcorner\Omega\urcorner \mid \lvert\Omega, \alpha\rvert$
labels	Γ, Δ, Ω	$::=$	$\alpha_1 \alpha_2 \ldots \alpha_n$

Backward: Where do I come from?



Name of the redex

$$\Omega \quad = \quad \alpha \;\; \Gamma \;\; \Delta_{pat} \;\; \Delta_{arg}$$

Pattern matching and history extraction

$$
\begin{array}{rcll}
\{\!\{ Y /_\theta \alpha : Z \}\!\}_m & = & \alpha : \{\!\{ Y /_\theta Z \}\!\}_m & m \in \{l, w\} \\
\{\!\{ Y /_\theta \hat{x} \}\!\}_l & = & (\epsilon, \{x \leftarrow Y\}) & x \in \theta \\
\{\!\{ Y /_\theta \hat{x} \}\!\}_w & = & (|Y|, \{x \leftarrow Y\}) & x \in \theta \\
\{\!\{ Y /_\theta ([\tau]p \to b) \}\!\}_l & = & (\epsilon, \bot) & \\
\{\!\{ Y /_\theta ([\tau]p \to b) \}\!\}_w & = & (|Y|, \bot) & \\
\{\!\{ \alpha : Z /_\theta X \}\!\}_m & = & \alpha : \{\!\{ Z /_\theta X \}\!\}_m & \text{matchability} \\
\{\!\{ \hat{x} /_\theta \hat{x} \}\!\}_m & = & (\epsilon, \{\}) & x \notin \theta \\
\{\!\{ A_1 A_2 /_\theta P_1 P_2 \}\!\}_m & = & \{\!\{ A_1 /_\theta P_1 \}\!\}_w \boxplus \{\!\{ A_2 /_\theta P_2 \}\!\}_l & \text{matchability} \\
\{\!\{ Y /_\theta X \}\!\}_m & = & (|Y||X|, \bot) & \text{matchability} \\
\{\!\{ Y /_\theta X \}\!\}_m & = & \text{undefined} & \text{otherwise}
\end{array}
$$

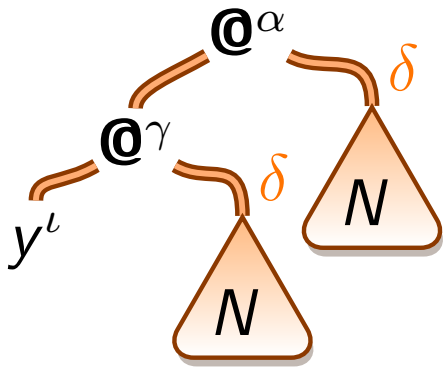
$$
\begin{array}{rcl}
|[\theta]P \to B| & = & \varepsilon \\
|\hat{x}| & = & \varepsilon \\
|T_1 T_2| & = & |T_1| \\
|\alpha : Z| & = & \alpha|Z|
\end{array}
$$

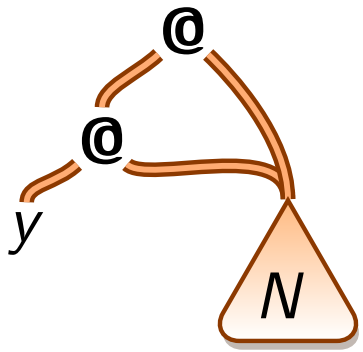
$\boxplus$	(Δ_2, σ_2)	$(\Delta_2, \bot)$	undef.
(Δ_1, σ_1)	$(\Delta_1\Delta_2, \sigma_1 \uplus \sigma_2)$	$(\Delta_1\Delta_2, \bot)$	undef.
$(\Delta_1, \bot)$	$(\Delta_1, \bot)$	$(\Delta_1, \bot)$	$(\Delta_1, \bot)$
undef.	undef.	undef.	undef.

From history to memory

From labelled terms to directed acyclic graphs.

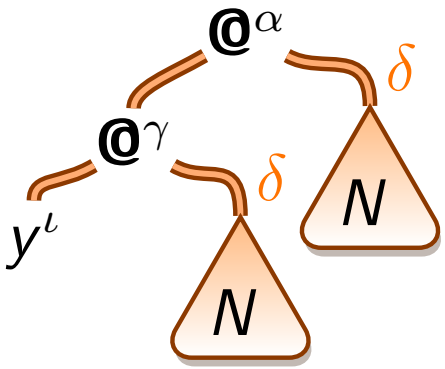
Labels as memory locations



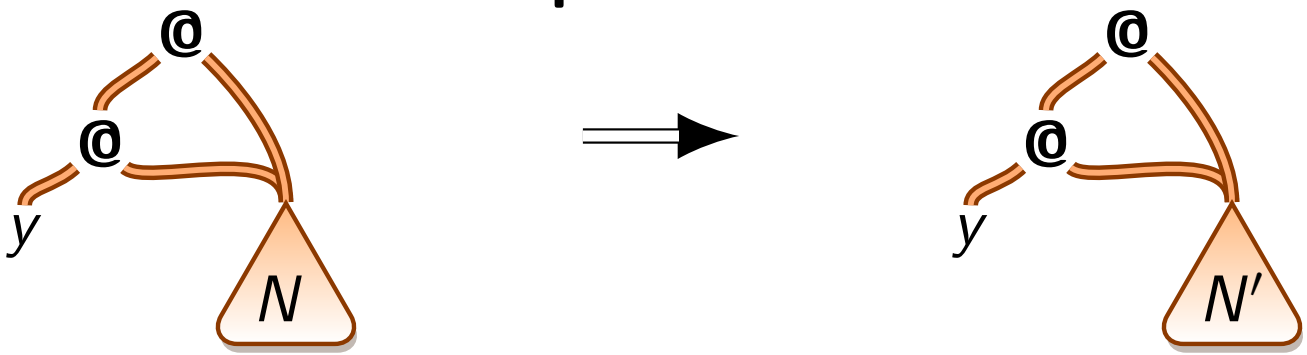


Sharing property: equal labels $\Rightarrow$ equal terms.

Labels as memory locations

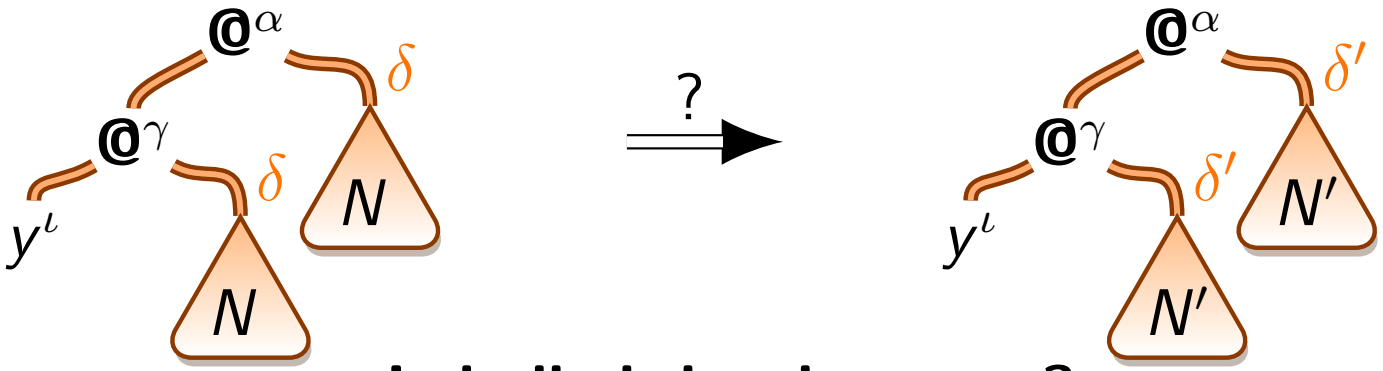


Graph reduction



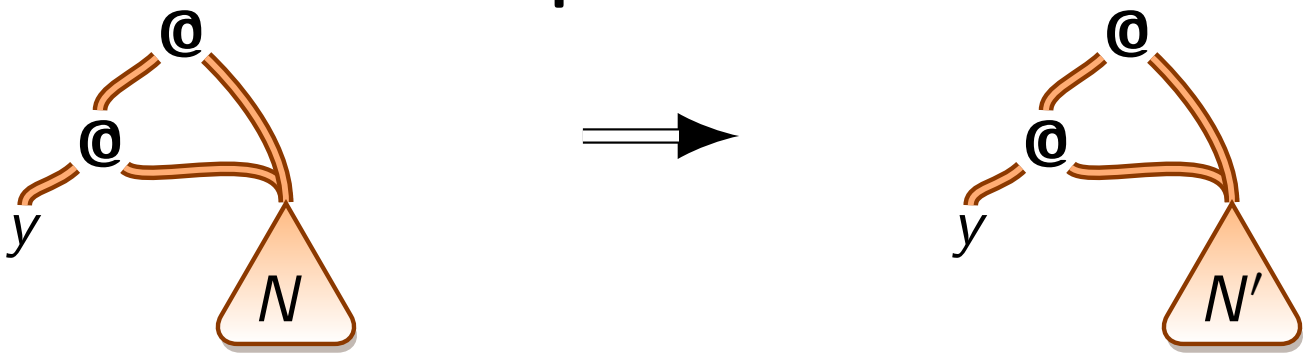
Sharing property: equal labels ⇒ equal terms.

Labels as memory locations



Labelled development?

Graph reduction



Sharing property: equal labels $\Rightarrow$ equal terms.

From labels to graph reduction

Complete labelled reduction:
development of all redexes with a given label.

Sharing theorem

Complete labelled reduction
preserves
the sharing property.

Hypothesis: start with initial labels.
Key: contribution is characterized by labels.

Conclusion

Present results

A causality analysis can yield

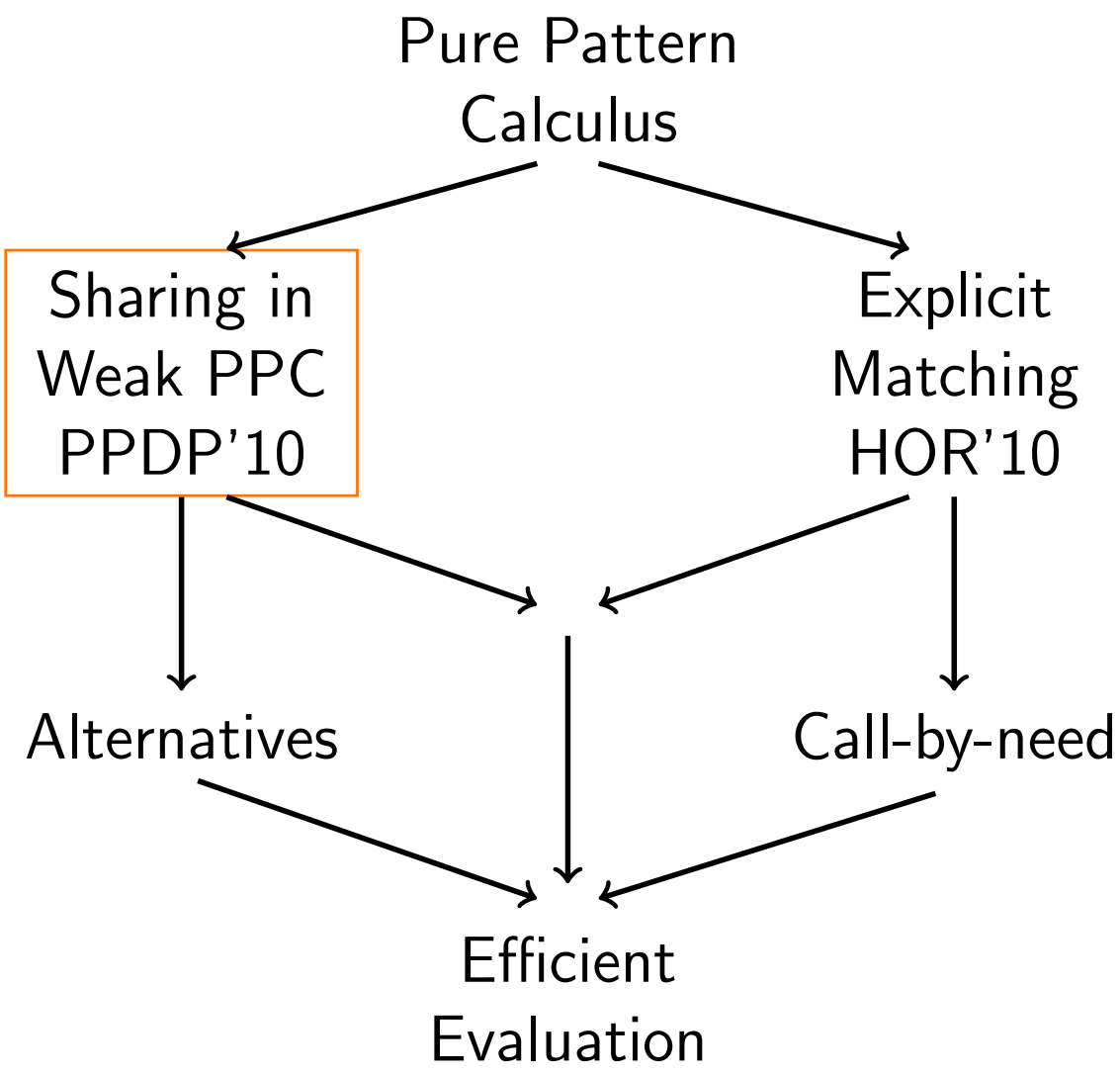
- a concrete DAG implementation.
- a proof of optimality of call-by-need.

Through PPC, this work reaches as well λ-calculus and pattern matching *à la ML*.

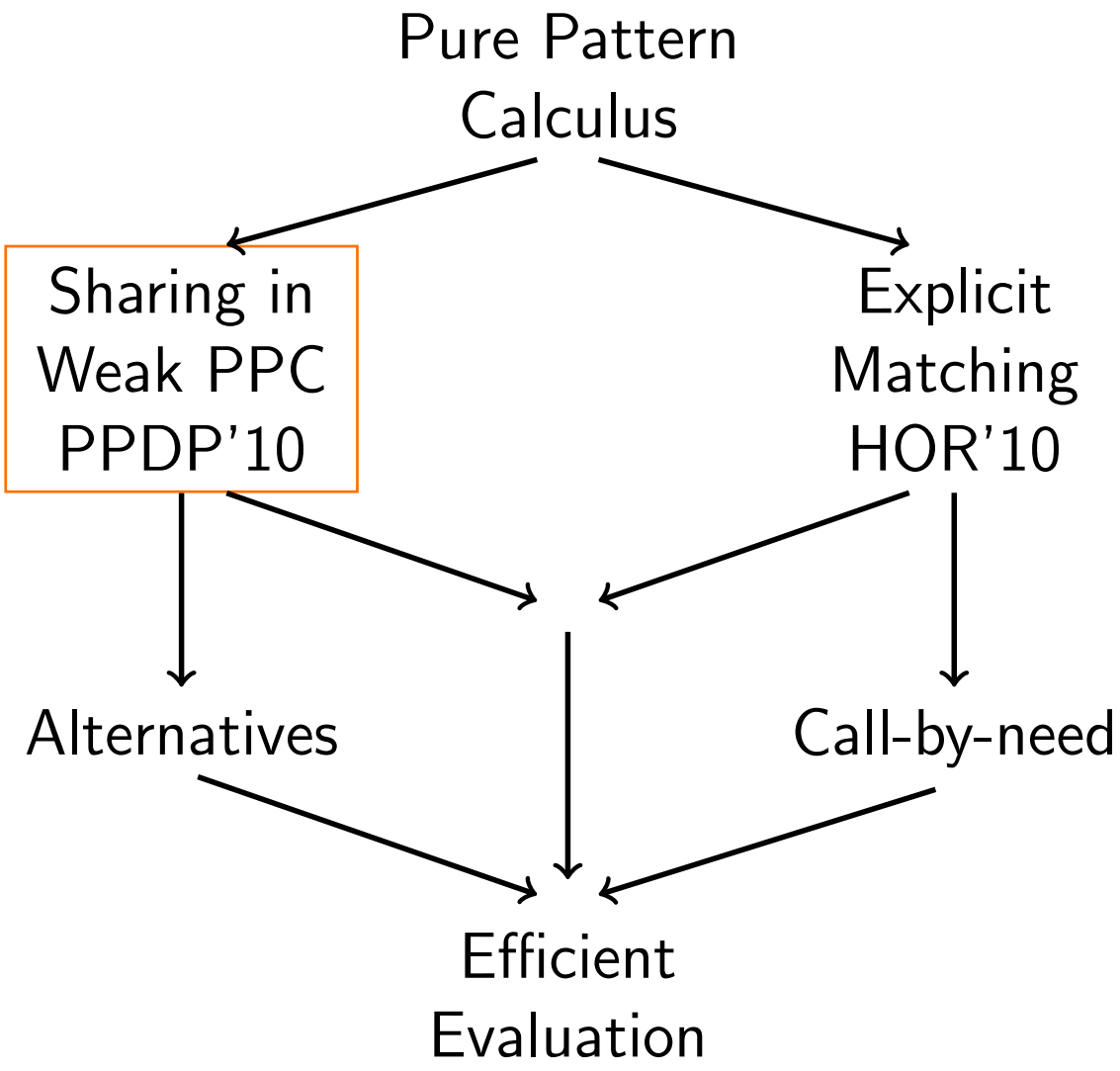
Future work

- Sharing in functions with multiple cases.
- Effective reduction strategy.

A broader picture



A broader picture



...and questions...