Representations in Syntax

Greg Kobele
Thomas Müntzer Lecture
Universität Leipzig
Introduction

Gereon (disappointedly)
everyone is moving to representations
Introduction

Gereon (disappointedly) everyone is moving to representations

What are representations?
- how should we think of them?
- what are the questions that we should ask?
- what is the trade-off with derivations?

What are derivations?
ibid.
We should focus on what information we need to support the interface maps
Representations of Derivations
A derivation

1. select every

2. select boy

3. merge 1 and 2 [DP every [NP boy]]

4. select laugh

5. merge 4 and 3 [VP laugh [DP every boy]]

6. select will

7. merge 6 and 5 [IP will [VP laugh [DP every boy]]]

8. move every boy [IP [DP every boy] [I′ will [VP laugh t]]]
A derivation

1. select every
every
A derivation

1. select every
   every
2. select boy
   boy

3. merge 1 and 2
   [DP every [NP boy]]

4. select laugh
   laugh

5. merge 4 and 3
   [VP laugh [DP every boy]]

6. select will
   will

7. merge 6 and 5
   [IP will [VP laugh [DP every boy]]]

8. move every boy
   [IP [DP every boy [IP [VP laugh t]]]]
A derivation

1. select every
   every
every

2. select boy
   boy

3. merge 1 and 2
   [DP every [NP boy]]
A derivation

1. select *every*
   *every* 
2. select *boy*
   *boy* 
3. merge 1 and 2
   
   
   [\textit{DP} *every* [\textit{NP} *boy*]]
4. select *laugh*
   *laugh*
1. select *every*  
   *every*  
2. select *boy*  
   *boy*  
3. merge 1 and 2  
   \[DP \text{ every } [NP \text{ boy }]\]  
4. select *laugh*  
   *laugh*  
5. merge 4 and 3  
   \[VP \text{ laugh } [DP \text{ every boy }]\]
A derivation

1. select every
every
2. select boy
boy
3. merge 1 and 2
   \[DP \text{ every } [NP \text{ boy }]]
4. select laugh
laugh
5. merge 4 and 3
   \[VP \text{ laugh } [DP \text{ every boy }]]
6. select will
will
A derivation

1. select every
   every
2. select boy
   boy
3. merge 1 and 2
   \[ DP \text{ every } [NP \text{ boy }] \]
4. select laugh
   laugh
5. merge 4 and 3
   \[ VP \text{ laugh } [DP \text{ every boy }] \]
6. select will
   will
7. merge 6 and 5
   \[ IP \text{ will } [VP \text{ laugh } [DP \text{ every boy }]] \]
A derivation

1. select every every
2. select boy boy
3. merge 1 and 2
   \[DP \text{ every } [NP \text{ boy }]\]
4. select laugh laugh
5. merge 4 and 3
   \[VP \text{ laugh } [DP \text{ every boy }]\]
6. select will will
7. merge 6 and 5
   \[IP \text{ will } [VP \text{ laugh } [DP \text{ every boy }]]\]
8. move every boy
   \[IP[DP \text{ every boy }][I' \text{ will } [VP \text{ laugh } t]]\]
Derivations are processes

A derivation is the process of constructing an expression

- derivations are important
- important things need to be thought about!
- it is helpful to be able to represent important things
Recipes are representations of processes

- **lexical items** are ingredients
- **merge and move** instead of bake, broil, whip, . . .
Derivations as recipes

1. select every
2. select boy
3. merge 1 and 2
4. select laugh
5. merge 4 and 3
6. select will
7. merge 6 and 5
8. move 3 in 7
Derivations are structured

Order is important

- Some things must happen before others
- Sometimes, it doesn’t matter

- merge det and noun
- *before* you merge the verb
- cream sugar and butter
- *before* you add the flour

Represent *before*-ness as dominance:
if A must happen *before* B, then B should be higher than A
Representing derivations

1. select
2. select
3. merge 1 and 2
4. select
5. merge 4 and 3
6. select
7. merge 6 and 5
8. move every boy
Representing derivations

1. select every
Representing derivations

1. select every
2. select boy
Representing derivations

1. select every
2. select boy
3. merge 1 and 2
Representing derivations

1. select every
2. select boy
3. merge 1 and 2
4. select laugh
Representing derivations

1. select every
2. select boy
3. merge 1 and 2
4. select laugh
5. merge 4 and 3
Representing derivations

1. select every
2. select boy
3. merge 1 and 2
4. select laugh
5. merge 4 and 3
6. select will
Representing derivations

1. select *every*
2. select *boy*
3. merge 1 and 2
4. select *laugh*
5. merge 4 and 3
6. select *will*
7. merge 6 and 5
Representing derivations

1. select every
2. select boy
3. merge 1 and 2
4. select laugh
5. merge 4 and 3
6. select will
7. merge 6 and 5
8. move every boy
The structure of derivations

- **move**
- **merge**
- **will**
- **merge**
- **laugh**
- **merge**
- **every**
- **boy**

**subtrees:** describe how to construct something
The structure of derivations

subtrees: describe how to construct something

x dominates y: to build x, you first have to build y
The structure of derivations

subtrees: describe how to construct something
x dominates y: to build x, you first have to build y
x c-commands y: before x can be used, you first have to build y
The structure of derivations

- **move**
- **merge**
- **will**
- **merge**
- **laugh**
- **merge**
  - **every**
  - **boy**

**subtrees**: describe how to construct something

- **x dominates y**: to build x, you first have to build y
- **x c-commands y**: before x can be used, you first have to build y
- **x and y are independent**: they can be built in any order
For comparison

1. cream sugar and butter
2. add eggs to 1
3. beat 2
4. add flour to 3
5. beat 4
6. stir chocolate chips into 5
7. bake 6
Infinite regress?

Do we have to build derivation trees?

NO!!!

- a recipe is a description of the process, not the process itself
- a recipe is helpful to think about what you did/will do

You can make a cookie without writing down what you did/are doing/will do
Properties of Derivations
Why do derivations look the way they do?

Why?
Why do derivations look the way they do?

Why?
because every selects for a N, and boy is an N.

```
move
  merge
    will
  merge
    merge
    laugh
    merge
    every
    boy
```
Why do derivations look the way they do?

Why?
because every selects for a $N$, and boy is an $N$
Why do derivations look the way they do?

Why?
because *laugh* selects for an *D*, and *every* is a *D*
Why do derivations look the way they do?

Why? 
because *laugh* selects for an *D*, and *every* is a *D*

![Diagram]

---

*move* 

*merge* 

*will* 

*laugh* 

*=d*

*merge* 

*every* 

*boy* 

*=n*

*n*

*d*
Why do derivations look the way they do?

Why?
because *will* selects for a *V*, and *laugh* is a *V*
Why do derivations look the way they do?

Why?
because *will* selects for a *V*, and *laugh* is a *V*
Why do derivations look the way they do?

Why?
because *every boy* needs case, and *will* assigns case
Why do derivations look the way they do?

Why?
because every boy needs case, and will assigns case
Derivations are endocentric
Derivations are endocentric
Derivations are endocentric
Derivations are endocentric
Derivations are endocentric

\[
\begin{align*}
\text{will} &= \text{v} + k \\
\text{laugh} &= \text{d} \\
\text{move} &= \text{v} \\
\text{every} &= \text{n} \\
\text{boy} &= \text{n} \\
\text{d} &= -k
\end{align*}
\]
Derivations are endocentric
(... unless countercyclicity)

1. select *laugh*
2. select *will*
3. merge 2 and 1
4. select *every*
5. select *boy*
6. merge 4 and 5
7. LATE merge 6 to 1 in 3
8. move 6 in 7
Headedness

will
merge
lugh
merge
every
boy
Headedness

```
will

laugh

merge

every

boy
```
Headedness

will

laugh

every

boy
Headedness

will

laugh

every

boy

15
Headedness

Headedness will laugh every boy
Headedness

every → boy

will → laugh
Headedness

evory

boy

will

laugh
The same recipe
Derived structure

every
every
Derived structure

every

every
Derived structure

every boy
every boy
Derived structure

merge
  /\   /
every  boy

DP
  /
\every  boy
Derived structure

laugh  merge
  every  boy

laugh  DP
  every  boy
Derived structure

merge
  
laugh
  
merge
  
  every
  
boy

VP
  
laugh
  
  every
  
boy

DP
Derived structure
Derived structure

```
 Derived structure

 T’
   /
  will
 /
VP
    /
   laugh
     /
   every
     /
boy

 Derived structure

 merge
 /
 will
 /
merge
 /
laugh
 /
merge
 /
every
 /
boy
```
Derived structure

move
  merge
    will
    merge
      laugh
      merge
        every
        boy

TP
  T'
    will
    VP
      laugh
      DP
        every
        boy
Derived structure (II)

every  every
Derived structure (II)

every

every
Derived structure (II)

every boy every boy
Derived structure (II)
Derived structure (II)

- laugh
  - merge
    - every
    - boy

- laugh
  - every
    - boy
Derived structure (II)

merge
\   \  
laugh merge
\     \  
every boy

laugh
\   \    
laugh every
\     \  
every boy
Derived structure (II)
 Derived structure (II)

merge
/   
will  merge
    /   
laugh  merge
    /     
every  boy

will
/   
laugh
/   
every
/   
boy
Derived structure (II)

![Diagram of derived structure](image)

- Derived structure (II)
  - move
  - merge
  - will
    - merge
    - laugh
    - every
      - boy
Derived structure (III)

every
every
Derived structure (III)

every every
Derived structure (III)

every boy  every boy
Derived structure (III)

```
merge
  /
 every  boy
```

```<
 every  boy
```
Derived structure (III)

laugh
  /   
merge    every
        /   
    boy

laugh
  /   
<    every
    /   
  boy
Derived structure (Ill)

merge
  \  
laugh    merge
  \     \      
every    boy

\<\ 

laugh
  \  
<
  \  
every    boy
Derived structure (III)

will

merge

laugh

every

boy

will

<

laugh

every

<

boy
Derived structure (III)

merge
  /     
will   merge
     /     
laugh merge
       /     
every boy

<
  /     
will <
     /     
laugh <
       /     
every boy
Derived structure (III)

```
move
  merge
    will
      merge
        every
          boy
          <i
          every
          boy
          will
          laugh
          t_i
```
Same or Different?

move
merge
will
merge
laugh
merge
every
boy

every
boy
will
laugh
t_i

will
every_i
will
laugh
every_i
boy_j
laugh
every_i
every_i
boy_j

t_P
T'
will
VP
laugh
DP
every
boy
Comparing Derived and Derivational Structure

- easy identity conditions for derivational structure
- derived structure is a copy of the derivation

Can we *replace* derived structure with derivational structure?

- what is at issue here?
Processing
Is derivational structure real or not?

Previously:

Do we have to build derivation trees?

NO!!

But now . . . ?

• am I proposing to replace derived trees w/ derivation trees?
• does this change things?
A parser

must construct a

1. well-formed
2. structure

the derivation

1. determines whether an expression is well-formed
2. gives you all the information you could ever want

a parser
A parser must construct a

1. well-formed
2. structure

the derivation

1. determines whether an expression is well-formed
2. gives you all the information you could ever want

a parser 1. must reconstruct a derivation
A parser

must construct a

1. well-formed
2. structure

the derivation

1. determines whether an expression is well-formed
2. gives you all the information you could ever want

a parser 1. must reconstruct a derivation and
2. needn’t reconstruct anything else
Parsing top down
Parsing top down

move
Parsing top down
Parsing top down

move

merge

every
Parsing top down

move

merge
every  boy
Parsing top down
Parsing top down

move

merge

will

merge

every

boy
Parsing top down
Parsing top down
Parsing top down

move
    |
merge
    |
will  merge
    |
laugh  merge
    |
every  boy
Looking at the parsing model

- parser must reconstruct the derivation
- so the derivation is a 'real' level of structure?
Compositionality
The question
how do we go from derivations to sounds and meanings?
Interpreting derivations

1. start w/ derivation tree
2. do the derivation described
3. interpret the derived object

But step 2. is just building a copy of what we started with!
Globality vs Locality

What is agreed upon?
never need to see the whole previous structure to decide about outcome of next step

'phases'
Compositionality
only use information about immediate arguments, and mode of combination, to determine result

\[
\begin{array}{c}
\text{merge} \\
\alpha \\
\beta
\end{array}
\] = f_{\text{merge}} \left[ \alpha \right] \left[ \beta \right]

if interface maps are compositional
• then we never need to construct a derivation tree
• can interpret every step as we postulate it
(an example is coming)
Ultra-locality

Compositionality only use information about immediate arguments, and mode of combination, to determine result

\[
\begin{array}{c}
\text{merge} \\
\alpha \\
\beta
\end{array}
\] = f_{merge} \[\alpha\] \[\beta\]

if interface maps are compositional

• then we never need to construct a derivation tree
• can interpret every step as we postulate it

(an example is coming)
The meaning of partial parse trees

\[ \lambda f \cdot \mathbb{[move]}(f \circ (\mathbb{[merge]} \circ \mathbb{[every]} \circ \mathbb{[boy]})) \]
Deforestation of parsing
Deforestation of parsing

\[ \lambda x, f \cdot (f \cdot x)' \]
Deforestation of parsing

\[ \lambda x, y, f \cdot (f \oplus (x \oplus y))' \]
Deforestation of parsing

\[ \lambda y \Box, f \Box (f \Box ([\text{every}] \oplus y \Box))' \]
Deforestation of parsing

\[ \lambda f \cdot (f \circ ([\text{every}] \oplus [\text{boy}]))' \]
Deforestation of parsing

\[ \lambda x_\Box, f_\Box. (x_\Box \oplus (f_\Box (\square \text{every} \oplus \square \text{boy})))' \]
Deforestation of parsing

\[ \lambda f. ([\text{will}] \oplus (f (\text{every} \oplus \text{boy}))))' \]
Deforestation of parsing

\[
\lambda x \Box, f \bigcirc. ([\textit{will}] \oplus (x \Box \oplus (f \bigcirc ([\textit{every}] \oplus [\textit{boy}])))))'
\]
Deforestation of parsing

\[ \lambda f.([\text{will}] \oplus ([\text{laugh}] \oplus (f ([\text{every}] \oplus [\text{boy}])))') \]
Deforestation of parsing

(move
  merge
    will
    merge
      laugh
      merge
        every
        boy)

\(([[will]] \oplus ([[laugh]] \oplus ([[every]] \oplus [boy])))'\)
A trick
add input structures to output domain

\[ f_{\text{merge}} \ a \ b = \begin{array}{c}
\text{merge} \\
\alpha \ \\
\beta
\end{array} \]
Dirty Tricks

A trick
add input structures to output domain

\[ f_{\text{merge}} \ a \ b = \begin{array}{c}
\text{merge} \\
\alpha \\
\beta
\end{array} \]

This is the point of derived structure
Compositionality is a restriction when

1. we limit what $f_{\text{merge}}$ and $f_{\text{move}}$ can do, and
2. we restrict what interpretations can be
Compositionality

Compositionality is a restriction when (Kracht)

1. we limit what $f_{\text{merge}}$ and $f_{\text{move}}$ can do, and
2. we restrict what interpretations can be

What should interpretations be?

- whatever we need
- if we end up needing craziness, we should worry
Computer scientists usually are happy to attach extra information to interpretations

- as long as it is finite
One man’s junk . . .

Computer scientists usually are happy to attach extra information to interpretations

- as long as it is finite

Example
add categorial information to strings
Computer scientists usually are happy to attach extra information to interpretations

- as long as it is finite

Example
add categorial information to strings

because we can think of this as being part of the operations instead:

not just merge, but merge-D-NP, merge-V-DP,...
What do we need

keep track of the unchecked syntactic features
(I won’t talk about this here)
What do we need

keep track of the unchecked syntactic features (I won’t talk about this here)

For PF
keep track of which phrases are still moving

but not of their internal structure
An example
An example

\[(\text{every, } n.d. - k)\]
An example

every boy

(every, =n.d.-k) (boy, n)
An example

\[
\text{merge} \\
\text{every} \quad \text{boy}
\]

\[
\frac{(\text{every}, =\text{n.d.} \cdot -k)}{(\text{every boy}, d. \cdot -k)} \\
(\text{boy}, n)
\]
An example

\[
\begin{align*}
\text{laugh} & \quad \text{merge} \\
\text{every} & \quad \text{boy} \\
\frac{(\text{every}, \text{boy}, d.-k)}{(\text{every}, \text{=n.d.-k})} & \quad (\text{boy, n}) \\
(laugh, =d.v) &
\end{align*}
\]
An example

```
(laugh, =d.v)  (every, =n.d.-k)  (boy, n)

(laugh, =d.v)  (every boy, d.-k)

(laugh, v), (every boy, -k)
```
An example

\[
\begin{array}{c}
\text{will} \\
\text{laugh} \\
\text{every} \\
\text{boy}
\end{array}
\]

\[
\frac{\text{will}}{\text{merge}} \quad \frac{\text{merge}}{\text{every}} \quad \frac{\text{boy}}{\text{laugh}}
\]

\[
\begin{array}{c}
\text{(will, } = v. + k.s) \\
\text{(laugh, } = d. v) \\
\text{(every boy, } d. - k) \\
\text{(every boy, } - k)
\end{array}
\]

\[
\text{(every, } = n.d. - k) \quad \text{(boy, } n)
\]
An example

\[
\begin{align*}
\text{will} & \quad \text{merge} \\
\text{laugh} & \quad \text{merge} \\
\text{every} & \quad \text{boy}
\end{align*}
\]

\[
\begin{align*}
\text{(will, } =v.+k.s) & \quad \text{(laugh, } =d.v) \quad \text{(every, } =n.d.-k) \quad \text{(every boy, } d.-k) \\
\text{(will laugh, } +k.s) & \quad \text{(laugh, } v) \quad \text{(every boy, } -k) \\
\end{align*}
\]
An example

\[
\text{(every boy will laugh, s)} \quad \frac{\text{(will laugh, } v\text{), (every boy, } -k\text{)}}{\text{(every boy, } d.-k\text{)}} \quad \frac{\text{(every, } =n.d.-k\text{)}}{\text{boy, n}}} 
\]
An example

```
(will, =v.+k.s)  
\(\text{move} \rightarrow \text{merge} \rightarrow \text{will} \rightarrow \text{merge} \rightarrow \text{laugh} \rightarrow \text{merge} \rightarrow \text{every} \rightarrow \text{boy}\)

\[
\begin{align*}
\text{(will laugh, +k.s), (every boy, -k)} & \quad \Rightarrow \quad \text{(every boy will laugh, s)} \\
\text{(every, =n.d.-k), (boy, n)} & \quad \Rightarrow \quad \text{(every boy, d.-k)} \\
\text{(laugh, =d.v)} & \quad \Rightarrow \quad \text{(every boy, -k)} \\
\text{(will, =v.+k.s)} & \quad \Rightarrow \quad \text{(laugh, v), (every boy, -k)}
\end{align*}
\]```
Semantics
What is necessary for semantics?

keep track of the unchecked syntactic features
(I won’t talk about this here)

For PF
keep track of which phrases are still moving
but not of their internal structure

For LF
???
What is the contribution of *praise every boy* to expressions it is part of?

Let's write instead:

\[
\text{every (boy)} \vdash \text{praise (x)}
\]
What is the contribution of *praise* every *boy* to expressions it is part of?

- A quantifier part: 
  
  $\text{every}(\text{boy})(\lambda x \ldots$ 

- And a property part: 
  
  $\text{praise}(x)$
What is the contribution of *praise every boy* to expressions it is part of?

- A quantifier part

  \[ \text{every(boy)}(\lambda x. \ldots) \]

- And a property part

  \[ \text{praise}(x) \]

Let’s write instead:

\[ [\text{every(boy)}]_x \vdash \text{praise}(x) \]
[\text{every(boy)}]_x \vdash \text{praise}(x)

The general case, with multiple stored quantifiers:

\[ [Q_1]_{x_1}, \ldots, [Q_i]_{x_i} \vdash M \]
Notation and Operations

\[ \text{[every(boy)]}_x \vdash \text{praise}(x) \]

The general case, with multiple stored quantifiers:

\[ [Q_1]_{x_1}, \ldots, [Q_i]_{x_i} \vdash M \]

The entire point is to ignore what is stored

\[
\frac{M}{\Gamma \vdash M} \uparrow \quad \frac{\Gamma \vdash M \quad \Delta \vdash N}{\Gamma, \Delta \vdash M \ N} \ <\ast> \]
Working with Storage

\[
\begin{align*}
\text{seem} & \quad \uparrow \quad \text{Pass} \\
\text{seem} & \quad \uparrow \quad \text{Pass} \\
\text{seem} & \quad \uparrow \quad \text{Pass} \\
\end{align*}
\]

\[
\begin{align*}
\text{Pass} & \quad \uparrow \quad \text{[every(boy)]}_x \vdash \text{praise}(x) \\
\text{[every(boy)]}_x & \vdash \text{Pass(praise}(x)) \\
\text{[every(boy)]}_x & \vdash \text{seem(Pass(praise}(x))) \\
\end{align*}
\]
Building *praise every boy*

\[
\begin{align*}
praise & \vdash praise \\
\end{align*}
\]

\[
\begin{align*}
\quad & \vdash every \\
\end{align*}
\]

\[
\begin{align*}
\quad & \vdash boy \\
\end{align*}
\]

\[
\begin{align*}
\quad & \vdash every \quad boy \\
\end{align*}
\]

*type mismatch!*
Building *praise every boy*

\[
\frac{\text{praise}}{\vdash \text{praise}} \quad \frac{\text{every}}{\vdash \text{every}} \quad \frac{\text{boy}}{\vdash \text{boy}}
\]
\[
\vdash \text{every boy} \quad <**>
\]

We want to 'insert a trace'

\[
\vdash M \\
[M]_x \vdash x
\]
Building *praise every boy*

We want to ’insert a trace’

\[
\frac{\vdash \text{praise}}{\vdash \text{praise}} \\
\frac{\vdash \text{every}}{\vdash \text{every}} \\
\frac{\vdash \text{boy}}{\vdash \text{boy}} <**>
\]

\[
\frac{\vdash \text{every}}{\vdash \text{every boy}} \\
\frac{[\text{every boy}]_x \vdash x}{x}
\]
Building *praise every boy*

\[
\text{praise} \quad \text{every} \quad \text{boy} \quad \text{praise} \quad \text{every} \quad \text{boy}.
\]

\[
\text{praise} \quad \text{every} \quad \text{boy} \quad [\text{every boy}]_x \quad \text{praise} \quad [\text{every boy}]_x.
\]

We want to 'insert a trace'

\[
\text{M} \quad \text{M} \quad \text{M}.
\]
Taking things out of storage

\[
\begin{align*}
\text{seem} & \quad \vdash \text{Pass} \\
\vdash \text{seem} & \quad \vdash \text{Pass} \\
\vdash [\text{every} (\text{boy})]_x & \vdash \text{praise} (x) \\
[\text{every} (\text{boy})]_x & \vdash \text{Pass} (\text{praise} (x)) \\
[\text{every} (\text{boy})]_x & \vdash \text{seem} (\text{Pass} (\text{praise} (x)))
\end{align*}
\]
Taking things out of storage

\[ \Gamma, [M_i]_{x_i}, \Delta \vdash N \]

\[ \frac{\Gamma, \Delta \vdash M_i \oplus (\lambda x_i. N)}{\langle \cdot \rangle_\oplus} \]
Taking things out of storage

\[
\frac{\text{Pass} \vdash \text{Pass}}{\vdash \text{Pass}} \quad \frac{[\text{every}(\text{boy})]_x \vdash \text{praise}(x)}{[\text{every}(\text{boy})]_x \vdash \text{Pass}(\text{praise}(x))} \quad [\text{every}(\text{boy})]_x \vdash \text{seem}(\text{Pass}(\text{praise}(x)))) 
\]  
\[\vdash \text{every}(\text{boy})(\lambda x.\text{seem}(\text{Pass}(\text{praise}(x)))) \quad \langle \cdot \rangle_{\mathcal{F}A}^1 \]

retrieval

\[
\frac{\Gamma, [M_i]_{x_i}, \Delta \vdash N}{\Gamma, \Delta \vdash M_i \oplus (\lambda x_i. N)} \quad \langle \cdot \rangle_\oplus^i
\]
Manipulating Stores

**pure**

\[
\frac{M}{\Gamma, \Delta \vdash M} \uparrow
\]

**apply**

\[
\frac{\Gamma \vdash M \quad \Delta \vdash N}{\Gamma, \Delta \vdash M \ N} \langle \ast \rangle
\]

**retrieve**

\[
\frac{\Gamma, [M_i]_{x_i}, \Delta \vdash N}{\Gamma, \Delta \vdash M_i \oplus (\lambda x_i. N)} \langle \cdot \rangle_i
\]

**store**

\[
\frac{\vdash M}{[M]_x \vdash x} \Box
\]
Understanding stores

\[[M_1]_{x_1}, \ldots, [M_i]_{x_i} \vdash N\]

\[\Rightarrow \lambda k. k \ M_1 \ \ldots \ M_i \ (\lambda x_1, \ldots, x_i. N)\]

Example

\[[\text{every boy}]_x \vdash \text{praise } x\]

\[\Rightarrow \lambda k. k \ (\text{every boy}) \ (\lambda x. \text{praise } x)\]
Some examples

pure

\[
\begin{align*}
\frac{M}{\Downarrow} & \quad \frac{M}{\Downarrow} \\
\end{align*}
\]

\[
\frac{M}{\lambda k. k \ M} \quad \frac{\lambda k. k \ M}{\lambda k. k \ M} \\
\]

\[
M \equiv \lambda k. k \ M
\]

storage

\[
\begin{align*}
\frac{\Downarrow}{\Downarrow} & \quad \frac{\lambda k. k \ M}{\lambda k. k \ M (\lambda x.x)} \\
\end{align*}
\]

\[
\begin{align*}
\frac{M}{\Downarrow} & \quad \frac{[M]_x \Downarrow x}{} \\
\end{align*}
\]

\[
\frac{\Downarrow}{\Downarrow} & \quad \frac{\lambda k. k \ M}{\lambda k. k \ M (\lambda x.x)} \\
\]

\[
\square m \equiv \lambda k. m (\lambda M. k \ M (\lambda x.x))
\]
More notation

idiom brackets

\[ (f \ a_1 \ldots a_i) \]

for \( f^{\uparrow} <*> a_1 <*> \ldots <*> a_i \)

application

Forward  \( f \triangleleft a := f \ a \)

Backward  \( a \triangleright f := f \ a \)
Minimalist semantics

\[
\begin{align*}
\text{[merge]} & \mapsto \lambda m, n. (|m \oplus n|) \\
\text{[merge]} & \mapsto \lambda m, n. (|m \oplus \square n|) \\
\text{[move]} & \mapsto \lambda m. m \\
\text{[move]} & \mapsto \lambda m. \langle m \rangle^k \\
\text{[ℓ]} & = \mathcal{I}(\ell)^\uparrow \\
\text{for} \quad \oplus \in \{\triangleleft, \triangleright\}
\end{align*}
\]
Unpacking the notation

Recall that

$$\lambda m, n. (m \triangleleft n)$$

means

$$\lambda m, n. (\triangleleft)^\uparrow \triangleright\triangleright m \triangleright\triangleright n$$

\[
\frac{\triangleleft}{\Gamma \vdash \triangleleft} \quad \frac{(m)}{\Gamma \vdash M \triangleleft} \quad \frac{(n)}{\Delta \vdash N} \quad \frac{\Delta \vdash N}{\Gamma, \Delta \vdash M \triangleleft N}
\]
Every boy laughs
Every boy laughs

\[
\begin{array}{c}
\text{move} \\
\downarrow \\
\text{merge} \\
\downarrow \\
\mathcal{I}(\text{will}) \\
\downarrow \\
\mathcal{I}(\text{laugh}) \\
\downarrow \\
\mathcal{I}(\text{every}) \\
\downarrow \\
\mathcal{I}(\text{boy})
\end{array}
\]
Every boy laughs
Every boy laughs

⊢ will

⊢ laugh \( \lambda m, n. (|m \triangle n|) \)

⊢ every

⊢ boy
Every boy laughs

⊢ l

⊢ m

├── m
├── l
│   │
│   └── will
│          └── m
├── l
└── l
    └── every boy
Every boy laughs

\[
\begin{align*}
\Gamma & \vdash \text{move} \\
\vdash \text{merge} \\
\Gamma & \vdash \text{will} \quad \lambda m, n. (|m \triangleleft \Box n|) \\
\vdash \text{laugh} & \quad \vdash \text{every boy}
\end{align*}
\]
Every boy laughs

⊢ will  [every boy]_x ⊢ laugh x
Every boy laughs

\[
\begin{array}{c}
\text{[[move]]} \\
\vdash \lambda m, n. (|m \triangleleft n|)
\end{array}
\]

\[\vdash \text{will} \quad \text{[every boy]}_x \vdash \text{laugh } x\]
Every boy laughs

$$[[\text{move}]$$

$$\vdash [\text{every boy}]_x \text{ will (laugh } x \text{)}$$
Every boy laughs

\[ \lambda m. \langle m \rangle_1 \]

\[ [\text{every boy}]_x \vdash \text{will (laugh } x) \]
Every boy laughs

\[ \vdash \text{every boy } (\lambda x.\text{will } (\text{laugh } x)) \]
Compositional interfaces

...allow for elimination of representational structure. Performance systems can 'use' the derivation in 'the wrong order' to construct the desired interface objects.
Deforestation of parsing (Again)

\[ \lambda x \, . \, x \]
Deforestation of parsing (Again)

\[
\lambda x_{\square}, f_{\bigcirc}. \langle f_{\bigcirc} \ x_{\square} \rangle^k
\]
Deforestation of parsing (Again)

\[ \lambda x \square, y \square, f \square \cdot \langle f \square \ (|| x \square \oplus y \square ||) \rangle_k \]
Deforestation of parsing (Again)

\[ \lambda y \Box, f \Box. \langle f \Box ((\| (\lambda z. \text{every} \oplus z)\uparrow y \Box)\|) \rangle^k_\oplus \]
Deforestation of parsing (Again)

\[ \lambda f_\circ \langle f_\circ (\text{every boy}^\uparrow) \rangle^k \]
Deforestation of parsing (Again)

\[ \lambda x_\Box, f_\bigcirc. \langle \langle x_\Box \oplus (f_\bigcirc (every\ boy^{\uparrow})) \rangle \rangle^k \]
Deforestation of parsing (Again)

\[
\lambda f. \langle \langle (\lambda z. \text{will} \oplus z)^\uparrow (f \circ (\text{every} \ \text{boy}^\uparrow)) \rangle \rangle ^k \oplus
\]
Deforestation of parsing (Again)

\[
\lambda x \Box, f \Box. \langle ((\lambda z. \text{will } \oplus z)^\dagger (x \Box \oplus (f \Box (\text{every boy}^\dagger)))) \rangle_\oplus^k
\]
Deforestation of parsing (Again)

\[
\lambda f \cdot \langle \langle (\lambda z. \text{will} \ (\text{laugh} \oplus z)) \uparrow (f \ (\text{every} \ \text{boy} \uparrow)) \rangle \rangle_{\oplus}^k
\]
Deforestation of parsing (Again)

\[
\text{every boy } \left( \lambda z. \text{will } (\text{laugh } z) \right)^\uparrow
\]
Conclusions

Derivations have structure

- with clear identity conditions
- of just the kind we want to assign

Interface maps
focus our attention on what matters:

   how much information (representation) we need to
compositionally interpret our derivations

Derived structure
is a familiar trick to circumvent compositionality