# 階層的グラフの戦略的書き換えによるプログラム実行モデリングとその利用

室屋 晃子

(京都大学 数理解析研究所)

早稲田大 上田研 セミナー, 30 January 2020

# Modelling program execution with *token-guided (hierarchical) graph rewriting*

Koko Muroya
(RIMS, Kyoto University)

# Overview: graphical models of program execution

graph rewriting

token passing

token-guided graph rewriting

applications:

- cost analysis
- language designs for programming with data-flow networks
- reasoning about observational equivalence
- visualising program execution

# Overview: graphical models of program execution

graph rewriting

token passing

token-guided graph rewriting

applications:

- cost analysis
- language designs for programming with data-flow networks
- reasoning about observational equivalence
- visualising program execution

# Graph-rewriting model

- dates back to [Wadsworth 1971]
- useful to achieve time-efficiency (by flexible sharing)
  - e.g. call-by-need evaluation without extra machinery

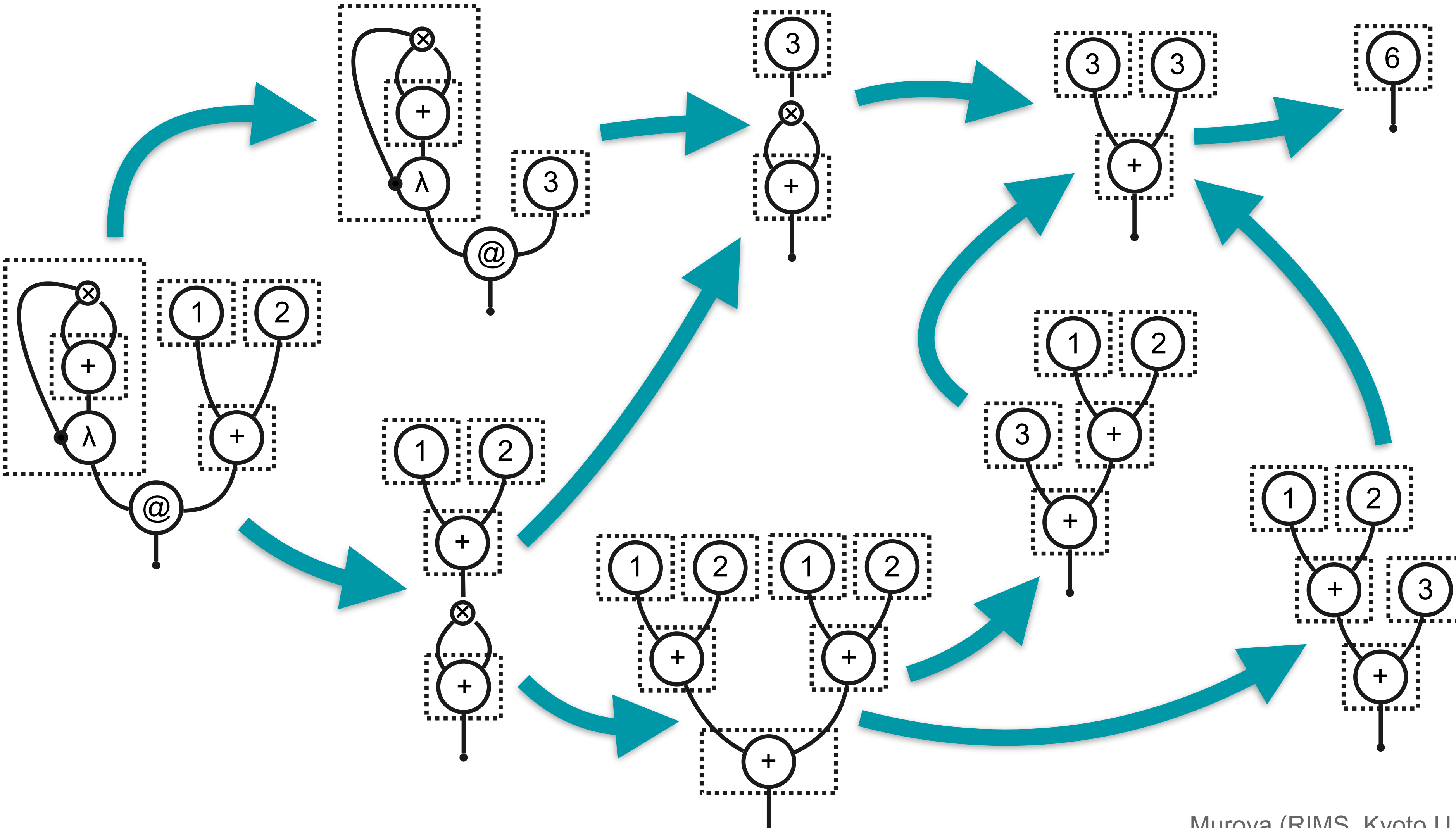
# Graph-rewriting model

program

`(λx. x + x) (1 + 2)`

result

6




Muroya (RIMS, Kyoto U.)

# Graph-rewriting model

- dates back to [Wadsworth 1971]
- useful to achieve time-efficiency (by flexible sharing)
  - e.g. call-by-need evaluation without extra machinery

<u>Question</u>

How to specify a strategy (i.e. a particular way of rewriting)?

# Overview: graphical models of program execution

graph rewriting

token passing

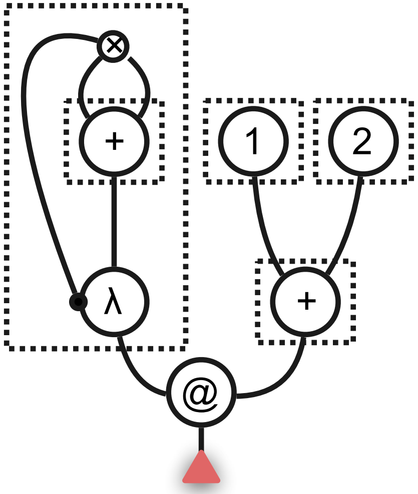
token-guided graph rewriting

applications:

- cost analysis
- language designs for programming with data-flow networks
- reasoning about observational equivalence
- visualising program execution

# Token-passing model

- based on *Geometry of Interaction* [Girard '89], pioneered by [Danos & Regnier '99] [Mackie '95]
- ingredients
  - the *token*, passed around on a fixed graph
  - *hierarchy* of the graph, managing re-evaluation



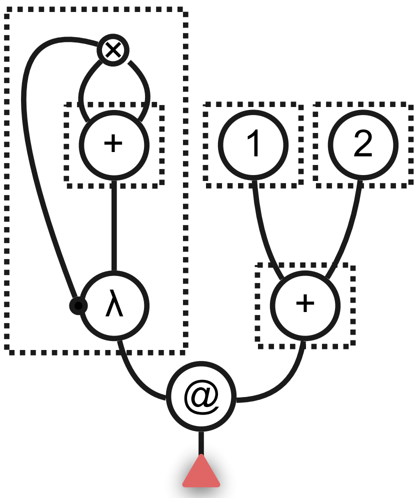
# Token-passing model

program

`(λx. x + x) (1 + 2)`

result

6



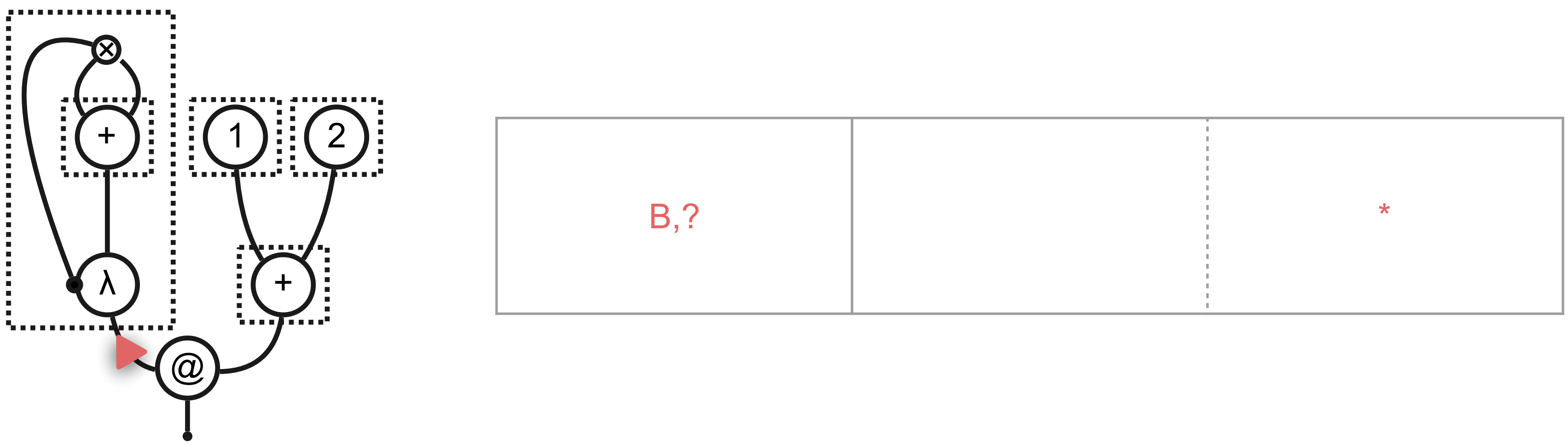

# Token-passing model

program

`(λx. x + x) (1 + 2)`

result

6



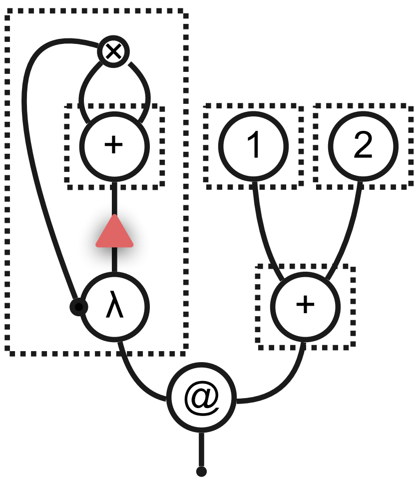
# Token-passing model

program

`(λx. x + x) (1 + 2)`

result

6




| | | |
|---|---|---|
| ? | * | * |

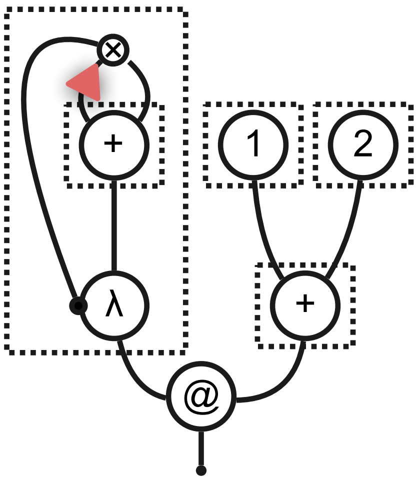
# Token-passing model

program

`(λx. x + x) (1 + 2)`

result

6




| | | |
|---|---|---|
| ? | * | <*,*> |

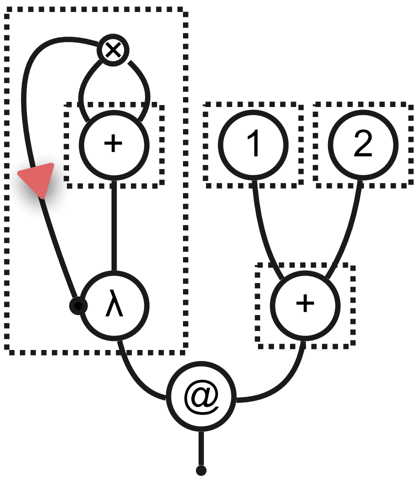
# Token-passing model

program

`(λx. x + x) (1 + 2)`

result

6




| | | |
|---|---|---|
| ? | * | L<*,*> |

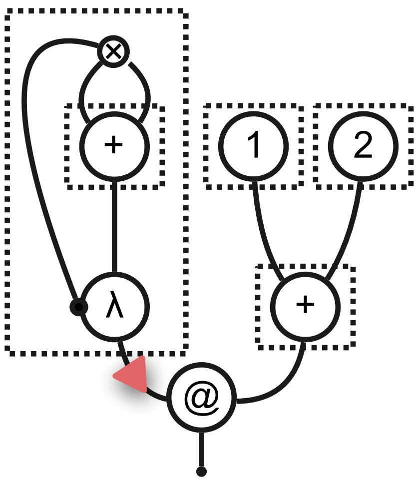
# Token-passing model

program

`(λx. x + x) (1 + 2)`

result

6



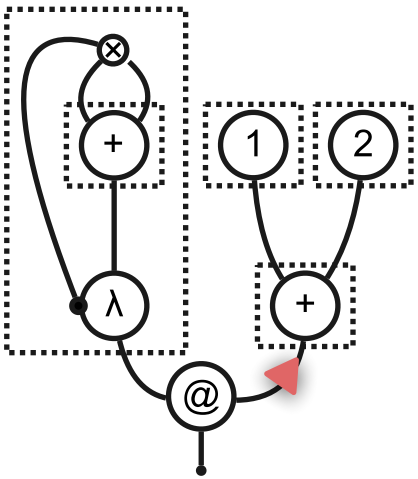

# Token-passing model

program

`(λx. x + x) (1 + 2)`

result

6



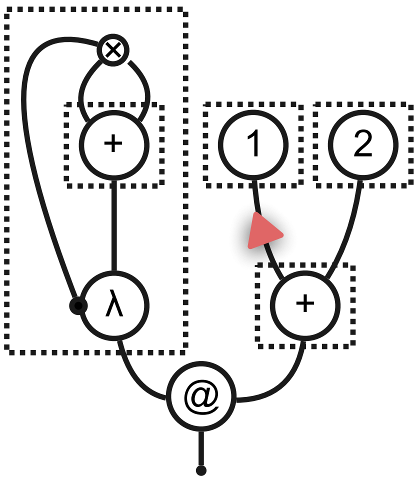

# Token-passing model

program

`(λx. x + x) (1 + 2)`

result

6



| ? | | <L<*,*>,*> |
|---|---|---|

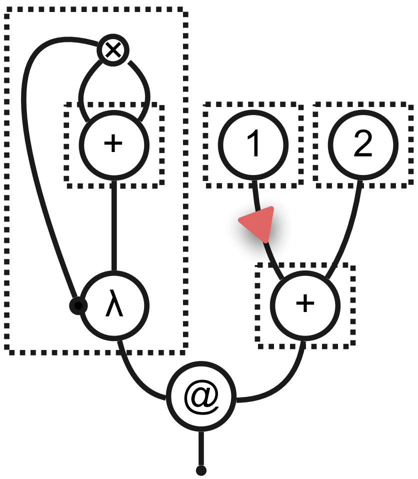
# Token-passing model

program

`(λx. x + x) (1 + 2)`

result

6




| | | |
|---|---|---|
| 1 | | <L<*,*>,*> |

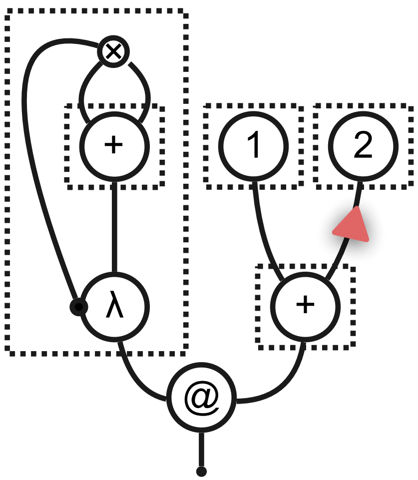
# Token-passing model

program

```
(λx. x + x) (1 + 2)
```

result

6




| | | |
|---|---|---|
| ? | | <L<*,*>,1> |

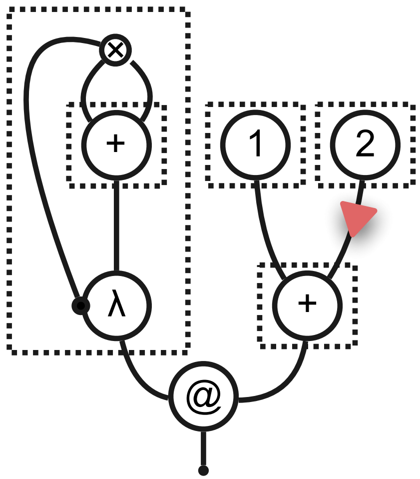
# Token-passing model

program

`(λx. x + x) (1 + 2)`

result

6




| | | |
|---|---|---|
| 2 | | <L<*,*>,1> |

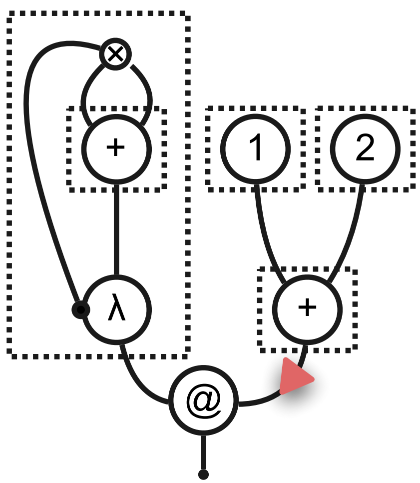
# Token-passing model

program

`(λx. x + x) (1 + 2)`

result

6




| | | |
|---|---|---|
| 3 | | L<*,*> |

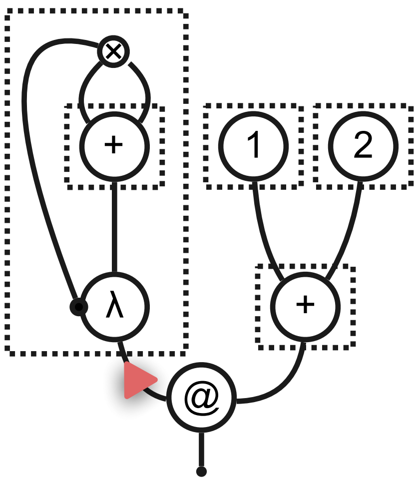
# Token-passing model

program

`(λx. x + x) (1 + 2)`

result

6




| | | |
|---|---|---|
| A,3 | | L<*,*> |

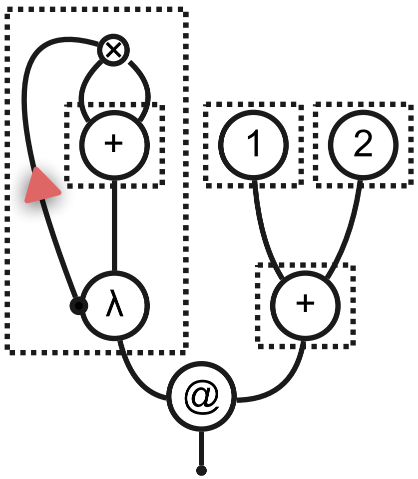
# Token-passing model

program

`(λx. x + x) (1 + 2)`

result

6




| | | |
|---|---|---|
| 3 | * | L<*,*> |

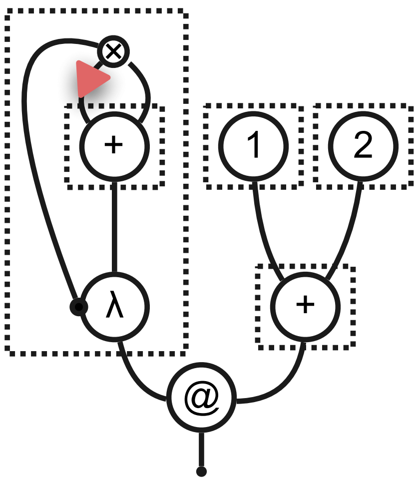
# Token-passing model

program

`(λx. x + x) (1 + 2)`

result

6




| | | |
|---|---|---|
| 3 | * | <*,*> |

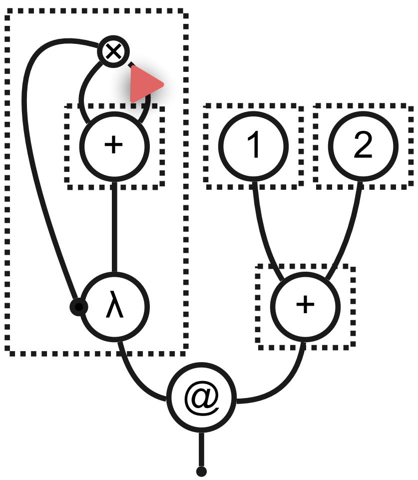
# Token-passing model

program

```
(λx. x + x) (1 + 2)
```

result

6




| | | |
|---|---|---|
| ? | * | <*,3> |

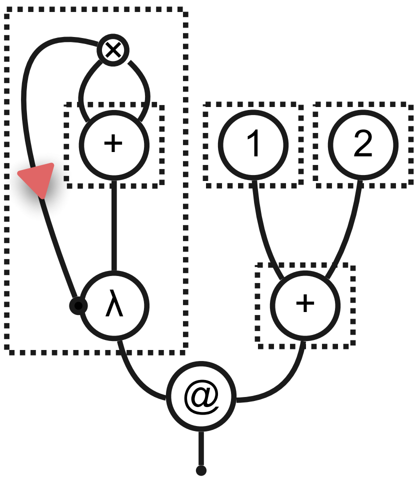
# Token-passing model

program

`(λx. x + x) (1 + 2)`

result

6




| | | |
|---|---|---|
| ? | * | L<*,3> |

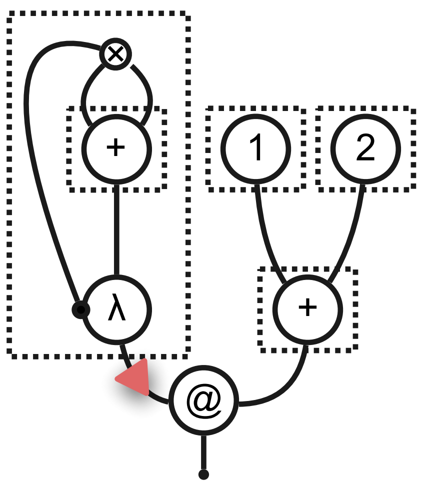
# Token-passing model

program

`(λx. x + x) (1 + 2)`

result

6




| | | |
|---|---|---|
| A,? | | L<*,3> |

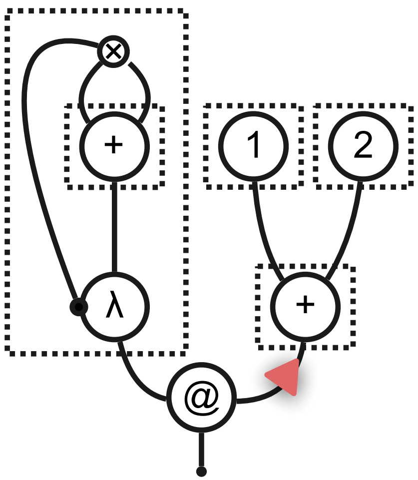
# Token-passing model

program

```
(λx. x + x) (1 + 2)
```

result

6




| | | |
|---|---|---|
| ? | | L<*,3> |

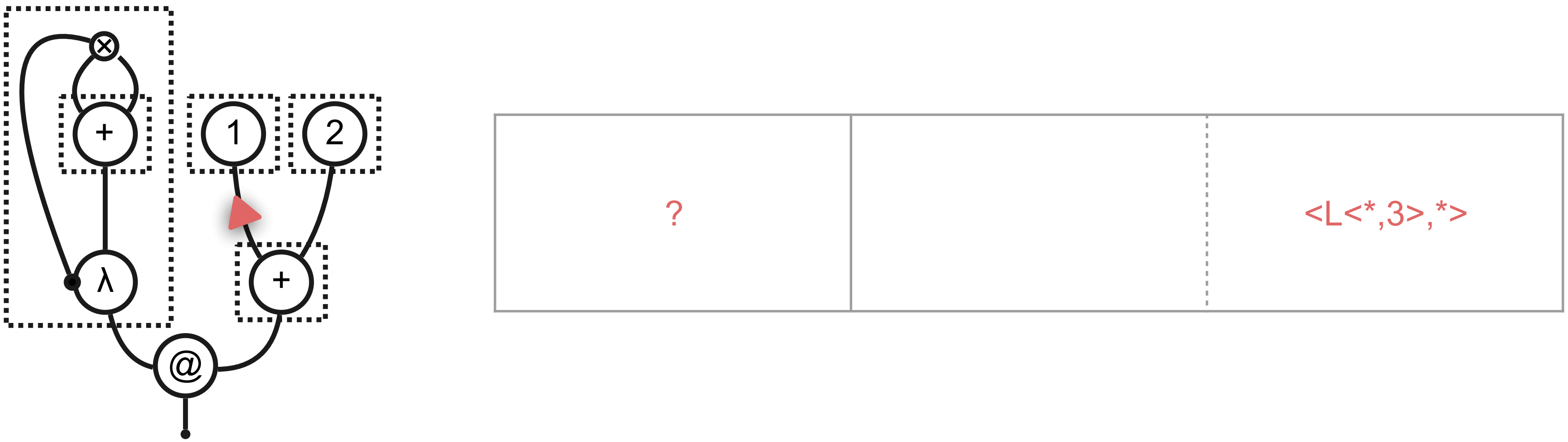
# Token-passing model

program

`(λx. x + x) (1 + 2)`

result

6



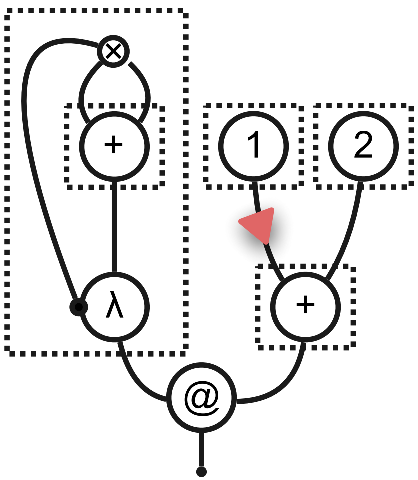
# Token-passing model

program

`(λx. x + x) (1 + 2)`

result

6




| | | |
|---|---|---|
| 1 | | <L<*,3>,*> |

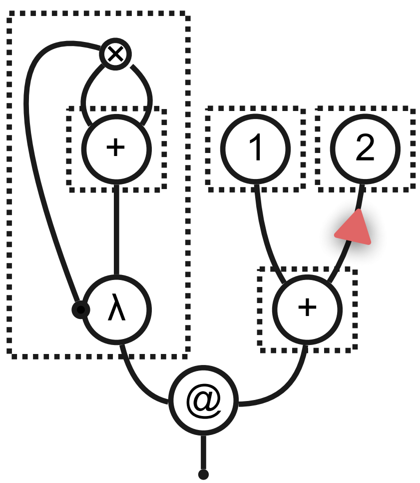
# Token-passing model

program

`(λx. x + x) (1 + 2)`

result

6




| | | |
|---|---|---|
| ? | | <L<*,3>,1> |

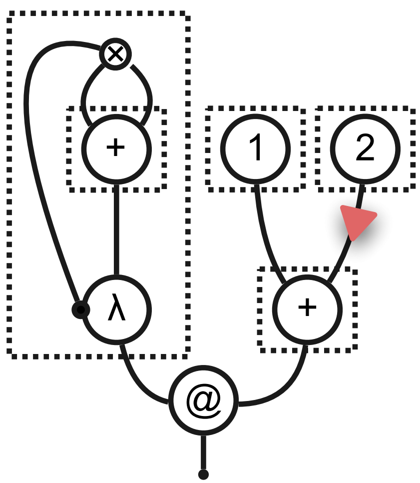
# Token-passing model

program

`(λx. x + x) (1 + 2)`

result

6




| | | |
|---|---|---|
| 2 | | <L<*,3>,1> |

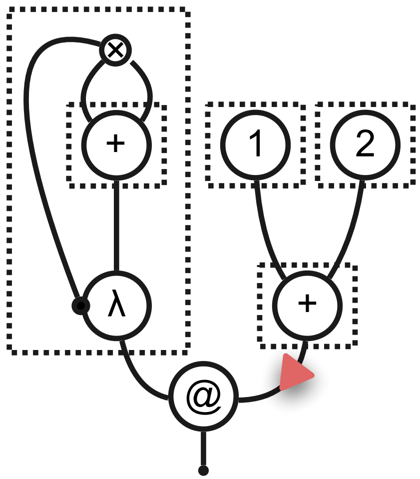
# Token-passing model

program

`(λx. x + x) (1 + 2)`

result

6




| | | |
|---|---|---|
| 3 | | L<*,3> |

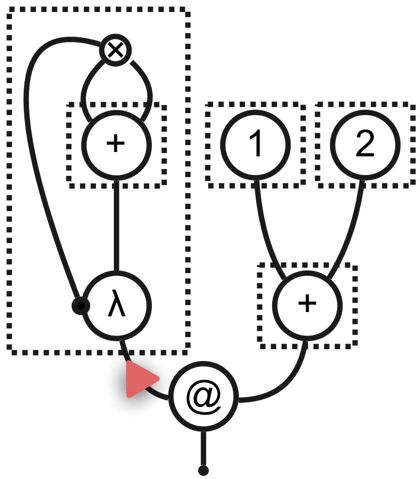
# Token-passing model

program

`(λx. x + x) (1 + 2)`

result

6




| | | |
|---|---|---|
| A,3 | | L<*,3> |

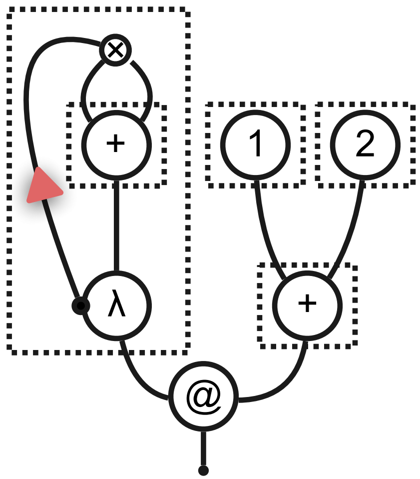
# Token-passing model

program

`(λx. x + x) (1 + 2)`

result

6




| | | |
|---|---|---|
| 3 | * | L<*,3> |

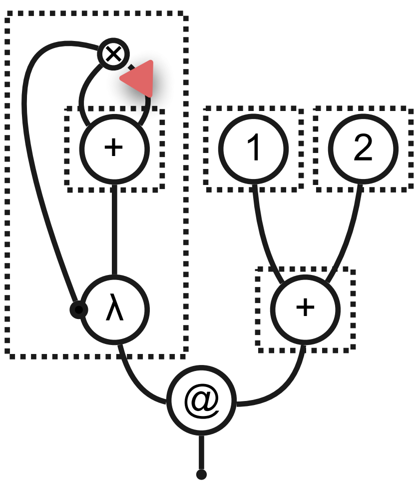
# Token-passing model

program

`(λx. x + x) (1 + 2)`

result

6




| | | |
|---|---|---|
| 3 | * | <*,3> |

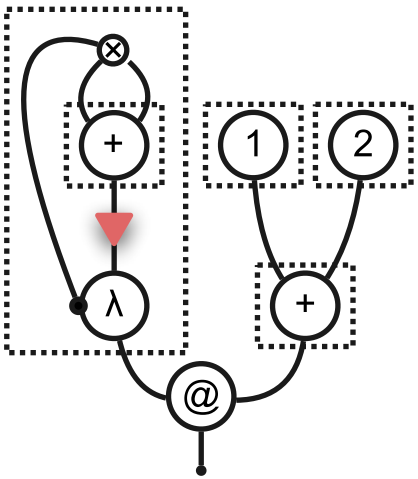
# Token-passing model

program

`(λx. x + x) (1 + 2)`

result

6




| | | |
|---|---|---|
| 6 | * | * |

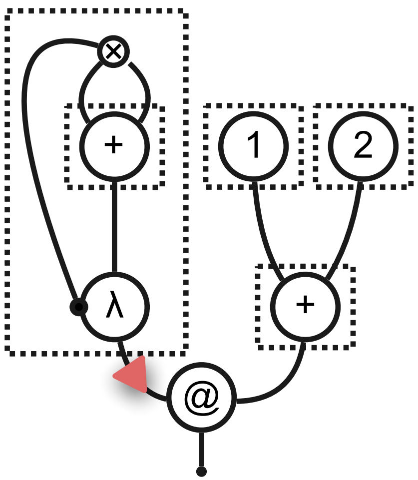
# Token-passing model

program

`(λx. x + x) (1 + 2)`

result

6




| | | |
|---|---|---|
| B,6 | | * |

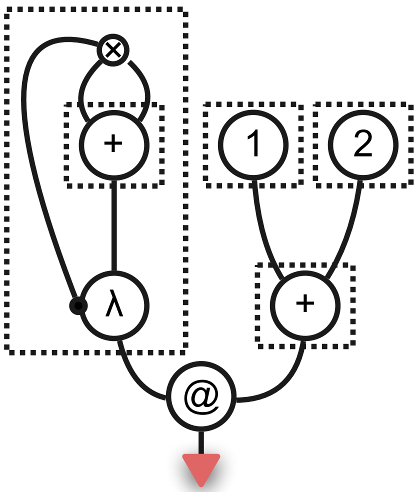
# Token-passing model

program

`(λx. x + x) (1 + 2)`

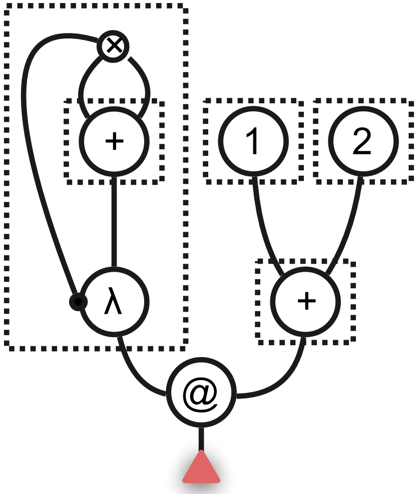
result

6




# Token-passing model

- based on *Geometry of Interaction* [Girard '89], pioneered by [Danos & Regnier '99] [Mackie '95]
- ingredients
  - the *token*, passed around on a fixed graph
  - *hierarchy* of the graph, managing re-evaluation
- said to be space-efficient (due to fixed graphs)
  - … but not really time-efficient (due to re-evaluation)




modelling call-by-name evaluation by default

Question

How to achieve time-efficiency?

# Models of program execution

graph rewriting

token passing

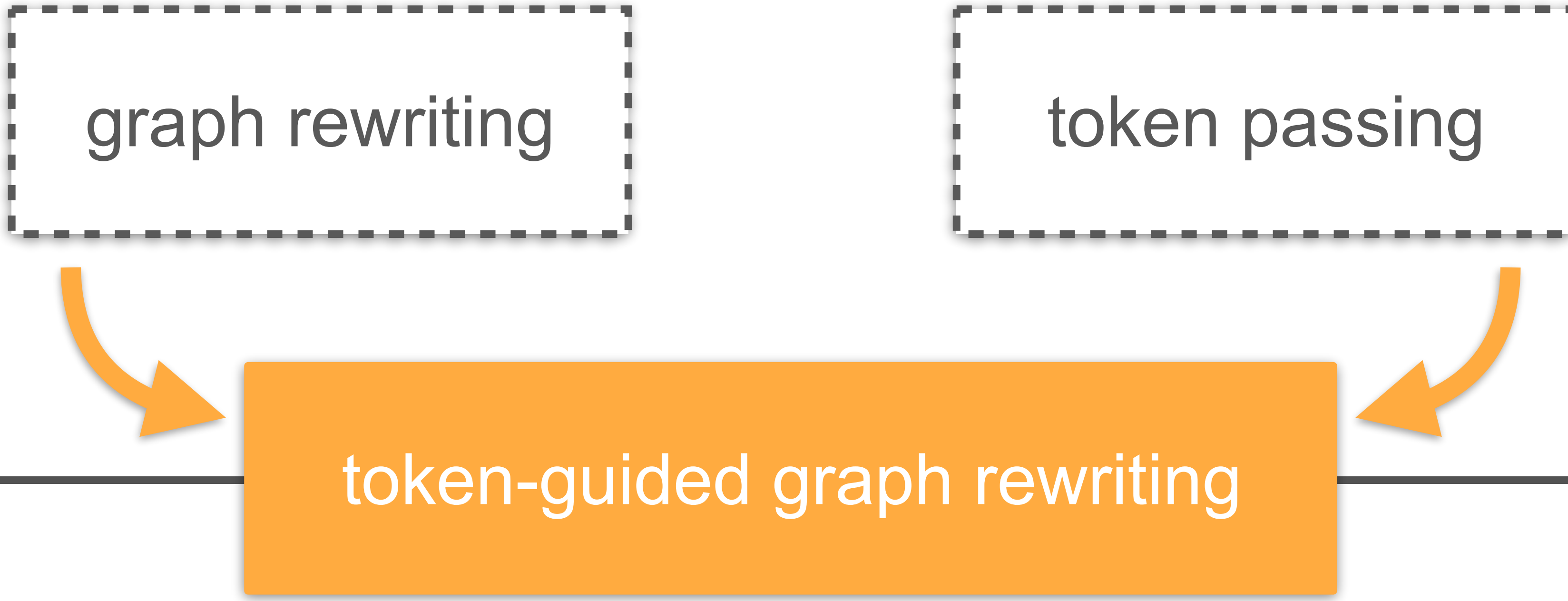
✔ time-efficiency

✔ space-efficiency

Questions

- a trade-off between time-efficiency and space-efficiency?
- a unified model to analyse the trade-off?

# Overview: graphical models of program execution




applications:

- cost analysis
- language designs for programming with data-flow networks
- reasoning about observational equivalence
- visualising program execution

# *Token-guided graph-rewriting model*

program

```
(λx. x + x) (1 + 2)
```

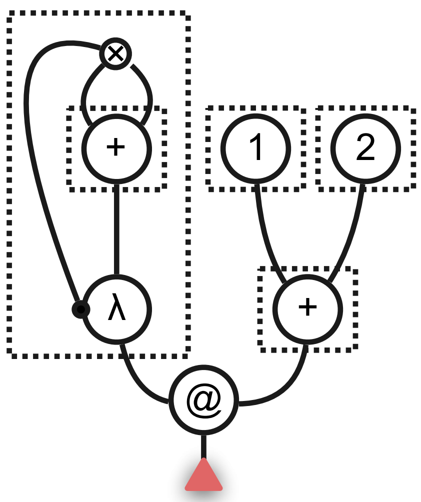
result

```
6
```

# Token-guided graph-rewriting model

program

```
(λx. x + x) (1 + 2)
```




result

6

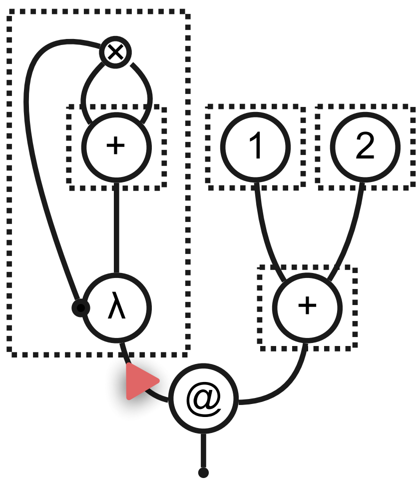
# Token-guided graph-rewriting model

program

`(λx. x + x) (1 + 2)`

result

6




| | | |
|---|---|---|
| B,? | | * |

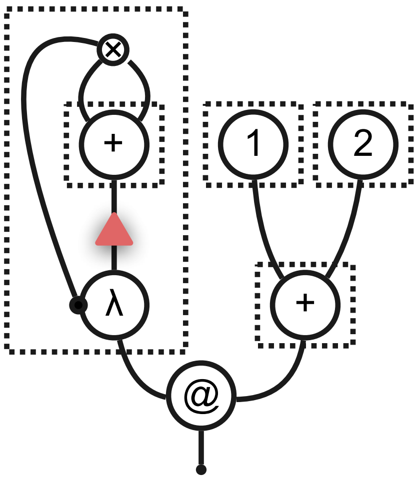
# Token-guided graph-rewriting model

program

`(λx. x + x) (1 + 2)`

result

6



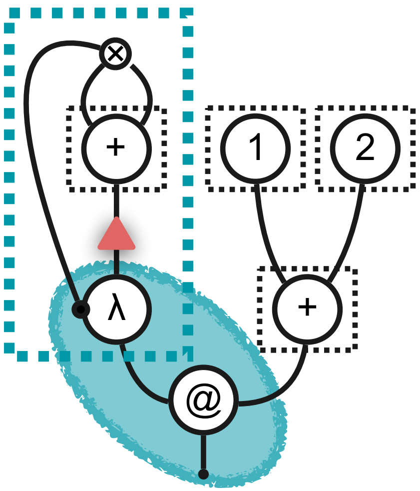

# Token-guided graph-rewriting model

program

`(λx. x + x) (1 + 2)`

result

6





the token has detected a redex…
> pass
> rewrite

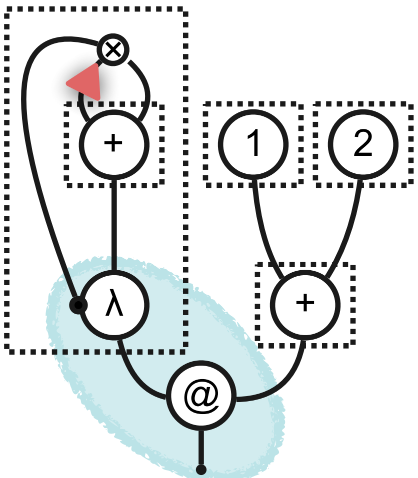
# Token-passing model

program

`(λx. x + x) (1 + 2)`

result

6




| | | |
|---|---|---|
| ? | * | <*,*> |

the token has detected a redex…

✔ **pass**

~~> rewrite~~

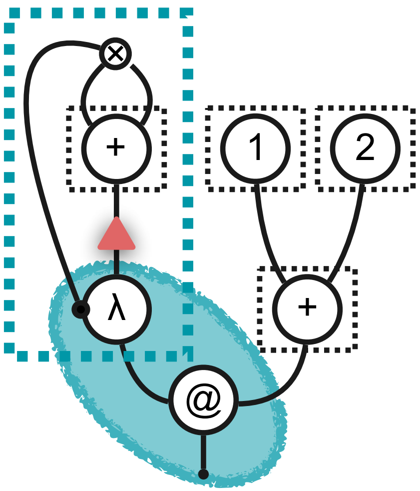
# Token-guided graph-rewriting model

program

`(λx. x + x) (1 + 2)`

result

6





the token has detected a redex…
> pass
> rewrite

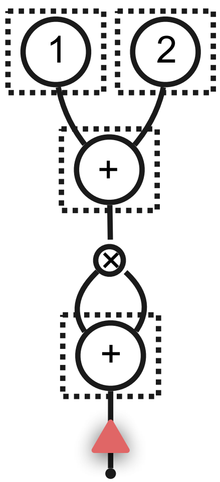
# *Token-guided graph-rewriting model*

program

`(λx. x + x) (1 + 2)`

result

6





the token has detected a redex…

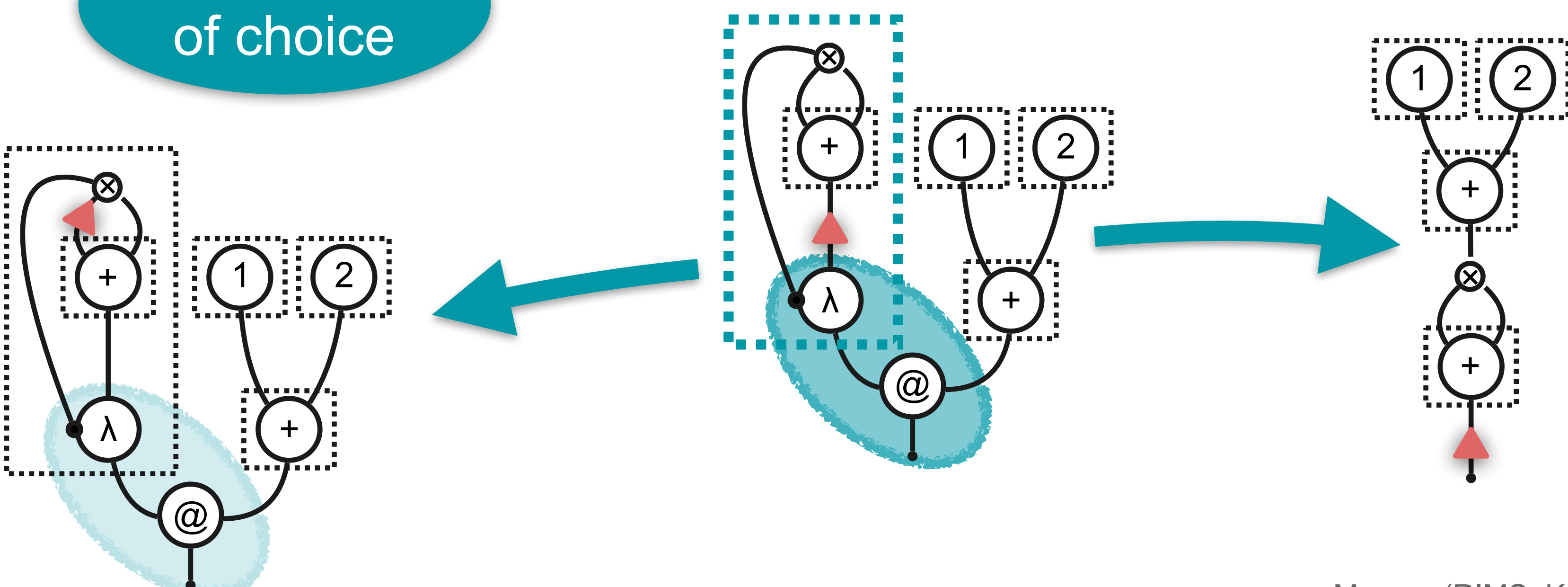
~~> pass~~

✔ **rewrite**

# *Token-guided graph-rewriting model*

- a combination of graph rewriting and token passing
- graph rewriting, *guided and controlled* by the token
  - redexes always *detected* by the token
  - rewrites can only be *triggered* by the token

freedom of choice



# *Modes* of token-guided graph-rewriting model

graph rewriting

token passing

"*maximum*" token-guided graph rewriting

rewrites triggered by the token *whenever possible*

modelling…

- by default: call-by-need evaluation
- also: call-by-value evaluation

  by changing the routing of the token

"*minimum*" token-guided graph rewriting

rewrites *never* triggered by the token

modelling…

- by default: call-by-name evaluation

# *Modes* of token-guided graph-rewriting model

graph rewriting

token passing

"*maximum*" token-guided graph rewriting

rewrites triggered by the token *whenever possible*

"*minimum*" token-guided graph rewriting

rewrites *never* triggered by the token

demo: https://koko-m.github.io/GoI-Visualiser/
for the (pure, untyped) lambda-calculus

# Overview: graphical models of program execution

graph rewriting

token passing

token-guided graph rewriting

applications:

- cost analysis
- language designs for programming with data-flow networks
- reasoning about observational equivalence
- visualising program execution

# Application 1: cost analysis

graph rewriting

✔ time-efficiency

token passing

✔ space-efficiency

Goal (also original motivation)

analysis of a trade-off between time-efficiency and space-efficiency

# Application 1: cost analysis

graph rewriting

"*maximum*" token-guided graph rewriting

rewrites triggered by the token *whenever possible*

token passing

"*minimum*" token-guided graph rewriting

rewrites *never* triggered by the token

**[— & Ghica, LMCS '19]**

**proof of *time-efficiency* of the "*maximum*" mode**

- call-by-need evaluation
- call-by-value evaluation

# Application 1: cost analysis

graph rewriting

token passing

"*maximum*" token-guided graph rewriting

rewrites triggered by the token *whenever possible*

"*minimum*" token-guided graph rewriting

rewrites *never* triggered by the token

**[ongoing work]**

**analysis of *various modes*, and hence the *time-space trade-off***

- "*maximum*" mode & "*minimum*" mode,
- "*up-to*" mode (e.g. allowing up to 100 rewrites),
- "*no-increase*" mode (i.e. forbidding growth of the graph), etc.

# Overview: models of program execution

graph rewriting

token passing

token-guided graph rewriting

applications:

- cost analysis
- language designs for programming with data-flow networks
- reasoning about observational equivalence
- visualising program execution

# Application 2: programming with data-flow networks

Goal programming language designs for:

- *construction* of a dataflow network
- *evaluation* of a dataflow network
- *update* of a dataflow network

# Application 2: programming with data-flow networks

Goal programming language designs for:

- *construction* of a dataflow network
- *evaluation* of a dataflow network
- *update* of a dataflow network

**[— & Cheung & Ghica, LICS '18] [Cheung & Darvariu & Ghica & — & Rowe, FLOPS '18]**

***Idealised TensorFlow***

# Application 2: programming with data-flow networks

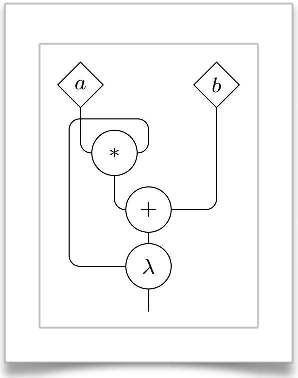
Goal  programming language designs for:

- *construction* of a dataflow network
- *evaluation* of a dataflow network
- *update* of a dataflow network

[— & Cheung & Ghica, LICS ’18] [Cheung & Darvariu & Ghica & — & Rowe, FLOPS ’18]

***Idealised TensorFlow***

- *construction* of a parametrised model
  (e.g. f(x) = a * x + b)
  as a network with **parameter nodes**



# Application 2: programming with data-flow networks

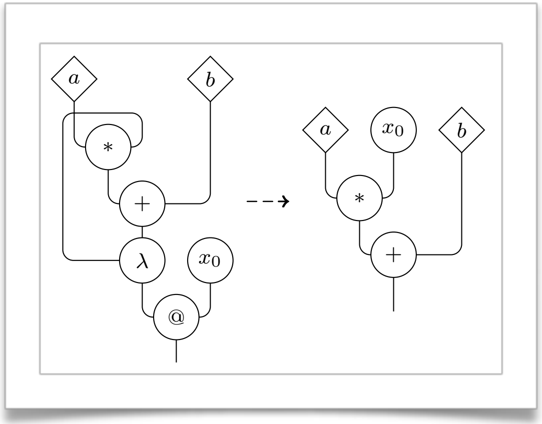
Goal  programming language designs for:

- *construction* of a dataflow network
- *evaluation* of a dataflow network
- *update* of a dataflow network

**[— & Cheung & Ghica, LICS '18] [Cheung & Darvariu & Ghica & — & Rowe, FLOPS '18]**

***Idealised TensorFlow***

- *prediction* with a parametrised model by
  1. **graph rewriting:**
     function application to input data



# Application 2: programming with data-flow networks

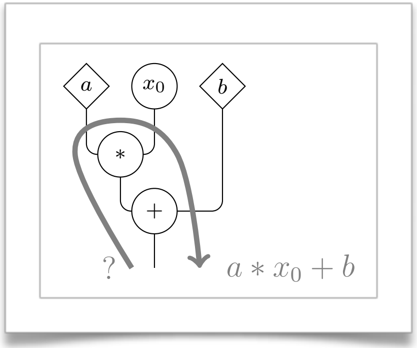
Goal programming language designs for:

- *construction* of a dataflow network
- *evaluation* of a dataflow network
- *update* of a dataflow network

**[— & Cheung & Ghica, LICS ’18] [Cheung & Darvariu & Ghica & — & Rowe, FLOPS ’18]**

***Idealised TensorFlow***

- *prediction* with a parametrised model by
  2. **token passing** over
  the resulting network



# Application 2: programming with data-flow networks

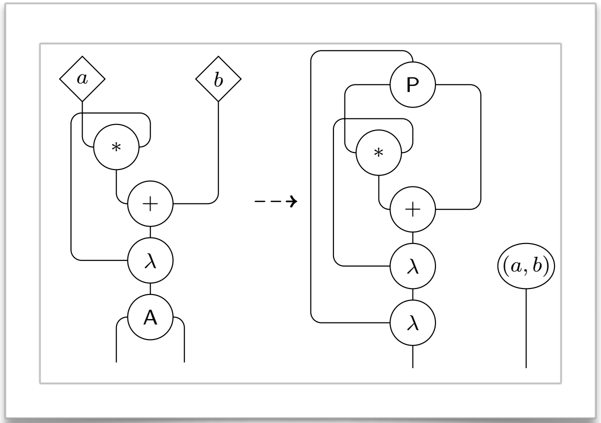
Goal programming language designs for:

- *construction* of a dataflow network
- *evaluation* of a dataflow network
- *update* of a dataflow network

**[— & Cheung & Ghica, LICS '18] [Cheung & Darvariu & Ghica & — & Rowe, FLOPS '18]**

***Idealised TensorFlow***

- *functional update* of parameters by
  1. **graph rewriting:**
     novel "*graph abstraction*"
     to turn a parametrised model
     into an ordinary function



# Application 2: programming with data-flow networks

Goal programming language designs for:

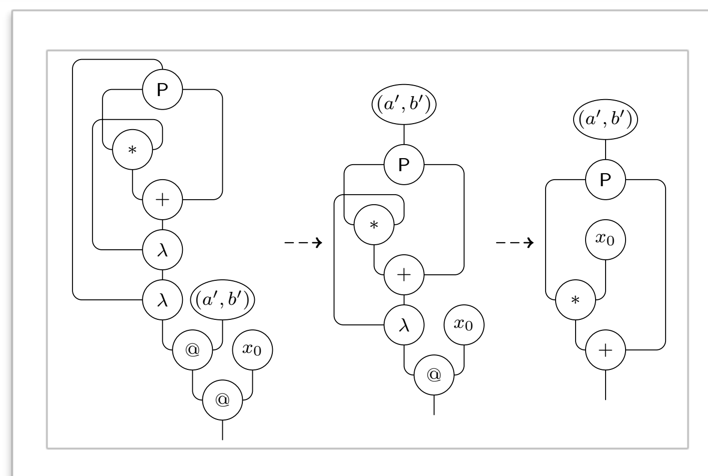
- *construction* of a dataflow network
- *evaluation* of a dataflow network
- *update* of a dataflow network

[— & Cheung & Ghica, LICS ’18] [Cheung & Darvariu & Ghica & — & Rowe, FLOPS ’18]

***Idealised TensorFlow***

- *functional update* of parameters by

  2. **graph rewriting:** function application to new parameter values



# Application 2: programming with data-flow networks

Goal programming language designs for:

- *construction* of a dataflow network
- *evaluation* of a dataflow network
- *update* of a dataflow network

**[— & Cheung & Ghica, LICS ’18] [Cheung & Darvariu & Ghica & — & Rowe, FLOPS ’18]**

***Idealised TensorFlow***

- extension of the simply-typed lambda-calculus with: *parameters*, “*graph abstraction*”, “*opaque*” vector types
- type soundness & some observational equivalences
- visualiser of token-guided graph rewriting
  https://cwtsteven.github.io/GoI-TF-Visualiser/CBV-with-CBN-embedding/index.html
- OCaml PPX implementation https://github.com/DecML/decml-ppx

# Application 2: programming with data-flow networks

Goal  programming language designs for:

- *construction* of a dataflow network
- *evaluation* of a dataflow network
- *update* of a dataflow network

for presentation,
see (esp. from 34:11):
https://www.youtube.com/watch?v=sgmpVedCsNM&t=102s

**[Cheung & Ghica & —, unpublished manuscript (arXiv:1910.09579)]**

***Transparent Synchronous Dataflow***

- extension of the simply-typed lambda-calculus with:
  *spreadsheet-like "cells"* (allowing circular dependency),
  "*step*" command  (updating cells step-by-step & concurrently)
- type soundness & some efficiency guarantee
- visualiser of token-guided graph rewriting https://cwtsteven.github.io/TSD-visual/
- OCaml PPX implementation https://github.com/cwtsteven/TSD
  (explained in https://danghica.blogspot.com/2019/11/making-ocaml-more-like-excel.html )

.)

# Overview: graphical models of program execution

graph rewriting

token passing

token-guided graph rewriting

applications:

- cost analysis
- language designs for programming with data-flow networks
- reasoning about observational equivalence
- visualising program execution

# Application 3: reasoning about observational equivalence

<u>Question(s)</u>

*Do two program fragments behave the same?*

or, *is it safe to replace a program fragment with another?*

```
let x = 100 in
let y = 50 in
y + y
```

$\xrightarrow{?}$

```
let y = 50 in
y + y
```

$\xrightarrow{?}$

```
50 + 50
```

```
let x = 100 in
let y = 50 in
y + y
```

$\xrightarrow{?}$

```
let x = 100 in
50 + 50
```

$\xrightarrow{?}$

```
50 + 50
```

if YES:

- justification of refactoring, compiler optimisation
- verification of programs

# Application 3: reasoning about observational equivalence

<u>Question(s)</u>

*Do two program fragments behave the same?*

# Application 3: reasoning about observational equivalence

<u>Question(s)</u>

~~*Do two*~~ *program fragments behave the same?*

***What*** *program fragments behave the same?*

the beta-law

$$(\lambda x\,.\,M)\,N \;\simeq\; M[x := N]$$

a parametricity law

$$\mathtt{let}\ a = \mathtt{ref}\ 1\ \mathtt{in}\ \lambda x\,.\,(a := 2; !a) \;\simeq\; \lambda x.2$$

# Application 3: reasoning about observational equivalence

<u>Question(s)</u>

~~*Do two*~~ *program fragments behave the same?*

***When do*** *program fragments behave the same?*

the beta-law

$$(\lambda x\,.\,M)\,N \;\simeq\; M[x := N]$$

Does the beta-law always hold?

# Application 3: reasoning about observational equivalence

Question(s)

~~*Do two*~~ *program fragments behave the same?*

***When do*** *program fragments behave the same?*

the beta-law

$$(\lambda x\,.\,M)\,N \;\simeq\; M[x := N]$$

Does the beta-law always hold?

**No**, it is violated by program contexts that can measure memory usage (e.g. with OCaml's Gc module)…

$$(\lambda x.0)\,100 \;\not\simeq\; 0$$

# Application 3: reasoning about observational equivalence

Question(s)

*Do two program fragments behave the same?*

***What fragments, in which contexts?***

… in the presence of (arbitrary) language features

pure vs. effectful (e.g. `50 + 50` vs. `ref 1`)

encoded vs. native (e.g. `State` vs. `ref`)

extrinsics (e.g. `Gc.stat`)

foreign language calls

# Application 3: reasoning about observational equivalence

Question(s)

*Do two **sub-graphs** behave the same?*

***What sub-graphs, in which contexts?***

… in *token-guided graph rewriting* for (arbitrary) language features

**[Ghica & — & Waugh Ambridge, unpublished manuscript (arXiv:1907.01257)]**

***Local reasoning for robust observational equivalence***

proof of (robustness of) observational equivalence

by exploiting **locality** of graph representation/syntax

# Application 3: reasoning about observational equivalence

**Locality** of graph syntax

"Does $\texttt{new}\ a \multimap 1\ \texttt{in}\ \lambda x.(a := 2;\ !a)$ behave the same as $\lambda x.2$?"

with linear syntax:

| | | | | | | |
|---|---|---|---|---|---|---|
| $\cdots$ | $\texttt{new}\ a \multimap 1\ \texttt{in}$ | $\cdots$ | $\lambda x.(a := 2;\ !a)$ | $\cdots$ | $\lambda x.(a := 2;\ !a)$ | $\cdots$ |

| | | | | |
|---|---|---|---|---|
| $\cdots$ | $\lambda x.2$ | $\cdots$ | $\lambda x.2$ | $\cdots$ |

# Application 3: reasoning about observational equivalence

**Locality** of graph syntax

"Does $\texttt{new}\ a \multimap 1\ \texttt{in}\ \lambda x.(a := 2;\ !a)$ behave the same as $\lambda x.2$?"

with linear syntax: ~~comparison between sub-terms~~

| | | | | | | |
|---|---|---|---|---|---|---|
| $\cdots$ | $\texttt{new}\ a \multimap 1\ \texttt{in}$ | $\cdots$ | $\lambda x.(a := 2;\ !a)$ | $\cdots$ | $\lambda x.(a := 2;\ !a)$ | $\cdots$ |

| | | | | |
|---|---|---|---|---|
| $\cdots$ | $\lambda x.2$ | $\cdots$ | $\lambda x.2$ | $\cdots$ |

# Application 3: reasoning about observational equivalence

**Locality** of graph syntax

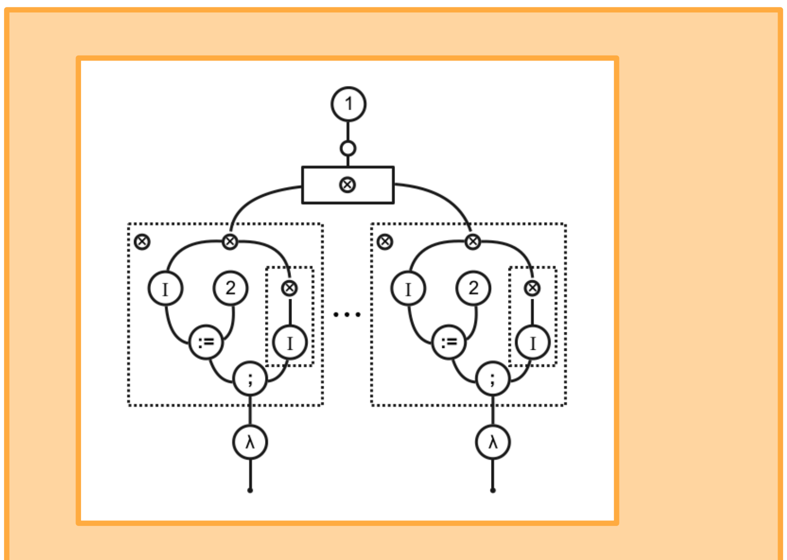
"Does $\texttt{new}\ a \multimap 1\ \texttt{in}\ \lambda x.(a := 2;\ !a)$ behave the same as $\lambda x.2$?"

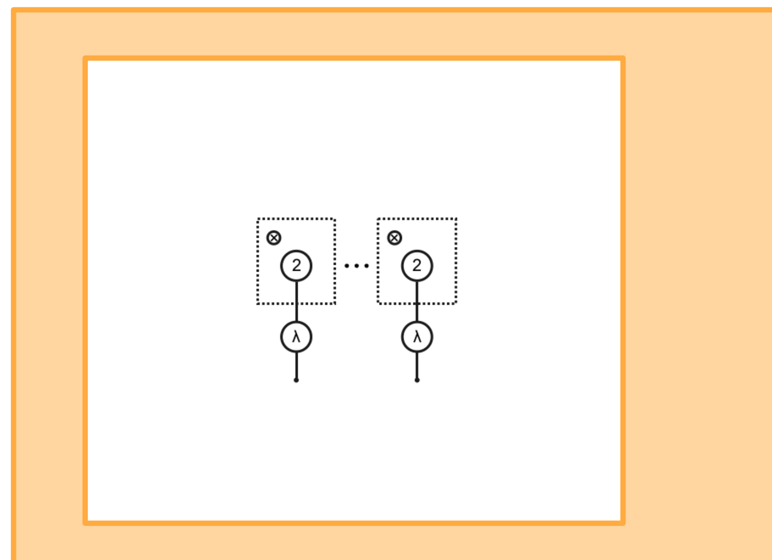
with linear syntax: ~~comparison between sub-terms~~

| | | | | | | |
|---|---|---|---|---|---|---|
| ... | $\texttt{new}\ a \multimap 1\ \texttt{in}$ | ... | $\lambda x.(a := 2;\ !a)$ | ... | $\lambda x.(a := 2;\ !a)$ | ... |

| | | | | |
|---|---|---|---|---|
| ... | $\lambda x.2$ | ... | $\lambda x.2$ | ... |

with graph syntax: comparison between sub-graphs



# Overview: graphical models of program execution

graph rewriting

token passing

token-guided graph rewriting

applications:

- cost analysis
- language designs for programming with data-flow networks
- reasoning about observational equivalence
- visualising program execution

# Application 4: visualising program execution

- OCaml Visual Debugger

  https://fyp.jackhughesweb.com/ by Jack Hughes

- comparison between programs
  - mutable state: encoded vs native

    https://www.youtube.com/watch?v=ysZdqocIu7E
  - sorting algorithms: insertion vs bubble

    https://www.youtube.com/watch?v=bZMSwo0zLio
  - sorting algorithms: merge vs insertion

    https://www.youtube.com/watch?v=U1NI-mWeNe0&t=213s

# Overview: graphical models of program execution

graph rewriting

token passing

token-guided graph rewriting

applications:

- cost analysis
- language designs for programming with data-flow networks
- reasoning about observational equivalence
- visualising program execution

# Overview: graphical models of program execution

graph rewriting

token passing

**token-guided graph rewriting**

biggest, persistent, **challenge**:

- **mathematical formalisation**
  - graph theory?
  - category theory? (DPO rewriting, string diagrams, …)
  - rewriting theory? (term-graph rewriting, …)