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)
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

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
Graph-rewriting model

Program: \((\lambda x. x + x) (1 + 2)\)

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

Question

How to specify a strategy (i.e. a particular way of rewriting)?
Overview: graphical models of program execution

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

- graph rewriting
- token passing
- token-guided graph rewriting
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:

$$(\lambda x. \; x + x) \; (1 + 2)$$

Result:

6
Token-passing model

program

\((\lambda x. x + x) (1 + 2)\)

result

\[ \boxed{6} \]
Token-passing model

program

\((\lambda x. x + x) (1 + 2)\)

result

6

Muroya (RIMS, Kyoto U.)
Token-passing model

Program:

\[(\lambda x. x + x)(1 + 2)\]

Result:

6
Token-passing model

Program

\[(\lambda x. x + x) \ (1 + 2)\]

Result

6

Diagram
Token-passing model

(program)

\[(\lambda x. x + x) (1 + 2)\]

(result)

6
Token-passing model

Program:

\[(\lambda x. x + x) (1 + 2)\]

Result:

6

Diagram:

\[
\begin{array}{c}
\lambda \downarrow \\
+ \downarrow \\
1 \downarrow \\
\times \downarrow \\
1 \downarrow \\
\times \downarrow \\
2 \downarrow \\
\downarrow \\
? \\
\end{array}
\]
Token-passing model

program

\((\lambda x. \ x + x) \ (1 + 2)\)

result

6

\(<\text{L}[^*,*],*>\)
Token-passing model

Program:

$$(\lambda x. \ x + x) \ (1 + 2)$$

Result:

6
Token-passing model

program

\(((\lambda x. x + x) \ (1 + 2))\)

result

6

<Muroya (RIMS, Kyoto U.)>
Token-passing model

Program: $$(\lambda x. x + x) \ (1 + 2)$$

Result: 6

Diagram:

Muroya (RIMS, Kyoto U.)
Token-passing model

program

\((\lambda x \cdot x + x) (1 + 2)\)

result

\[ 6 \]

Muroya (RIMS, Kyoto U.)
Token-passing model

program

\((\lambda x. \ x + x) \ (1 + 2)\)

result

\[ A,3 \]

\[ L<*,*> \]
Token-passing model

Program:

\[(\lambda x. x + x) (1 + 2)\]

Result:

6
Token-passing model

Program: 

\((\lambda x. x + x) \ (1 + 2)\)

Result: 

6
Token-passing model

Program:
\[(\lambda x. x + x) \ (1 + 2)\]

Result:
6

Diagram:

\[
\begin{array}{c}
? \\
\times \\
+ \\
\lambda \at \ 1 \\
\ + \\
1 \\
\ 2
\end{array}
\]

\[
\begin{array}{c}
* \\
<*,3>
\end{array}
\]
Token-passing model

program

\((\lambda x. \ x + x) \ (1 + 2)\)

result

6
Token-passing model

program

\((\lambda x. x + x) \ (1 + 2)\)

result

6
Token-passing model

program: \((\lambda x. \ x + x) \ (1 + 2)\)

result: 6
Token-passing model

\[(\lambda x. x + x)(1 + 2)\]

\(\lambda x\)

\(+\)

\(1\)

\(2\)

\(?\)

\(<L<*,3>,*>\)
Token-passing model

Program:

\[ (\lambda x. x + x) \ (1 + 2) \]

Result:

\[ 6 \]

Muroya (RIMS, Kyoto U.)
Token-passing model

program

\[(\lambda x. x + x) (1 + 2)\]

result

6

[Muroya (RIMS, Kyoto U.)]
Token-passing model

program

$$(\lambda x. x + x) \ (1 + 2)$$

result

6

$$(\lambda \@ \times \@ 1 \times 2)\ <L<*,3>,1>$$
Token-passing model

program

\((\lambda x. x + x) \ (1 + 2)\)

result

6
Token-passing model

Program: \((\lambda x. x + x) (1 + 2)\)

Result: 6

Diagram: [Diagram of token-passing model]
Token-passing model

Program:

$$(\lambda x. \ x + x) \ (1 + 2)$$

Result:

$$6$$
Token-passing model

(program)

\[(\lambda x. x + x) (1 + 2)\]

(result)

6

3 * <*,3>
Token-passing model

Program:

\[(\lambda x. x + x) \ (1 + 2)\]

Result:

6
Token-passing model

Program:

\[(\lambda x. \ x + x) \ (1 + 2)\]

Result:

6
Token-passing model

Program:

\((\lambda x. x + x) \ (1 + 2)\)

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)

**Question**

How to achieve time-efficiency?
Models of program execution

- **graph rewriting**
  - ✔ time-efficiency

- **token passing**
  - ✔ 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

graph rewriting

token passing

token-guided graph rewriting
Token-guided graph-rewriting model

program

$$\lambda x. x + x \ (1 + 2)$$

result

6
Token-guided graph-rewriting model

program

\[(\lambda x. x + x) (1 + 2)\]

result

\[6\]
Token-guided graph-rewriting model

$\lambda x. x + x \ (1 + 2)$

Result: 6
Token-guided graph-rewriting model

Program:

\[(\lambda x. x + x) (1 + 2)\]

Result:

6
Token-guided graph-rewriting model

program

\[(\lambda x. \ x + x) \ (1 + 2)\]

result

6

the token has detected a redex…
> pass
> rewrite
Token-passing model

The token has detected a redex…

✓ pass

> rewrite

Program

\[(\lambda x. x + x) (1 + 2)\]

Result

6
Token-guided graph-rewriting model

(program)  \((\lambda x. x + x) (1 + 2)\)  

(result)  6  

the token has detected a redex...  
> pass  
> rewrite
Token-guided graph-rewriting model

Program: \((\lambda x. x + x) (1 + 2)\)

Result: 6

The token has detected a redex...
> pass
✓ rewrite

Muroya (RIMS, Kyoto U.)
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**
  - "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

- **token passing**
  - "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**
  - "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

**demo**: [https://koko-m.github.io/GoI-Visualiser/](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

Goal (also original motivation)

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

- ✔ time-efficiency
- ✔ space-efficiency

graph rewriting

token passing
Application 1: cost analysis

- **Graph Rewriting**
  - "maximum" token-guided graph rewriting
    - rewrites triggered by the token *whenever possible*
  - "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**
  - “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

token-guided graph rewriting

graph rewriting

token passing

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 & Darvari & Ghica & — & Rowe, FLOPS ’18]

*Idealised TensorFlow*

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

Muroya (RIMS, Kyoto U.)
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 & Darvari & 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
- 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

[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/](https://cwtsteven.github.io/TSD-visual/)

- OCaml PPX implementation [https://github.com/cwtsteven/TSD](https://github.com/cwtsteven/TSD)

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

Do two program fragments behave the same?

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

if YES:

- justification of refactoring, compiler optimisation
- verification of programs
Application 3: reasoning about observational equivalence

Question(s)

_Do two program fragments behave the same?_
Applicatio n 3: reasoning about observational equivalence

Question(s)

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

let $a = \text{ref } 1$ in $\lambda x. (a := 2; !a) \simeq \lambda x. 2$
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?
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
Question(s)

Do two *sub-graphs* behave the same?

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

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

[Giha & — & 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 `new a → 1 in λx.(a := 2; !a)` behave the same as `λx.2`?”

with linear syntax:

```
  ... new a → 1 in ... λx.(a := 2; !a) ... λx.(a := 2; !a) ...
  ... λx.2 ... λx.2 ...
```
Application 3: reasoning about observational equivalence

**Locality** of graph syntax

“Does `new a ↠ 1 in λx.(a := 2; !a)` behave the same as `λx.2`?”

with linear syntax: comparison between sub-terms

```
⋯ new a ↠ 1 in ⋯ λx.(a := 2; !a) ⋯ λx.(a := 2; !a) ⋯
⋯                   ⋯ λx.2                   ⋯ λx.2 ⋯
```
Application 3: reasoning about observational equivalence

**Locality** of graph syntax

“Does $\text{new } a \rightarrow 1 \text{ in } \lambda x. (a := 2; !a)$ behave the same as $\lambda x. 2$?”

with linear syntax: comparison between sub-terms

$$\cdots \text{new } a \rightarrow 1 \text{ in } \cdots \lambda x. (a := 2; !a) \cdots \lambda x. (a := 2; !a) \cdots$$

$$\cdots \lambda x. 2 \cdots \lambda x. 2 \cdots$$

with graph syntax: comparison between sub-graphs

Muroya (RIMS, Kyoto U. & U. B’ham.)
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=ysZdqoclu7E

  - 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

biggest, persistent, **challenge**:

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