Token-passing semantics *with and without* rewriting

Koko Muroya
(RIMS, Kyoto University
& University of Birmingham)

Steven W. T. Cheung
Dan R. Ghica
(University of Birmingham)

CRECOGI meeting 2018 (Paris), 08 October 2018

Token-passing semantics *without* rewriting

Token-passing semantics *without* rewriting

execution models, provided by Girard's GoI,

of functional programming

- call-by-name λ-calculus [Danos&Regnier '99] [Mackie '95]
- call-by-value λ-calculus [Fernandez&Mackie '02]
- and more: PCF, effects, concurrency...

Token-passing semantics *without* rewriting

execution models, provided by Girard's GoI,

of functional programming

- call-by-name λ-calculus [Danos&Regnier '99] [Mackie '95]
- call-by-value λ-calculus [Fernandez&Mac
- and more: PCF, effects, concurrency...

(MELL) proof net, λ-graph, ...

program	*diagram*
evaluation	*token passing*

Token-passing semantics *without* rewriting

program	*diagram*
evaluation	*token passing*
result	*token data/position*

live demo:

Jamping Abstract Machine for call-by-name λ-calculus [DR99]

https://koko-m.github.io/GoI-Visualiser/

Token-passing semantics *without* rewriting

program	***λ-graph + !-boxes***
evaluation	*token passing* ***+ jumping***
result	*token ~~data~~/position*

live demo:

Jamping Abstract Machine for call-by-name λ-calculus [DR99]

https://koko-m.github.io/GoI-Visualiser/

Token-passing semantics *without* rewriting

program	*diagram*
evaluation	*token passing*
result	*token data/position*

execution models of functional programming

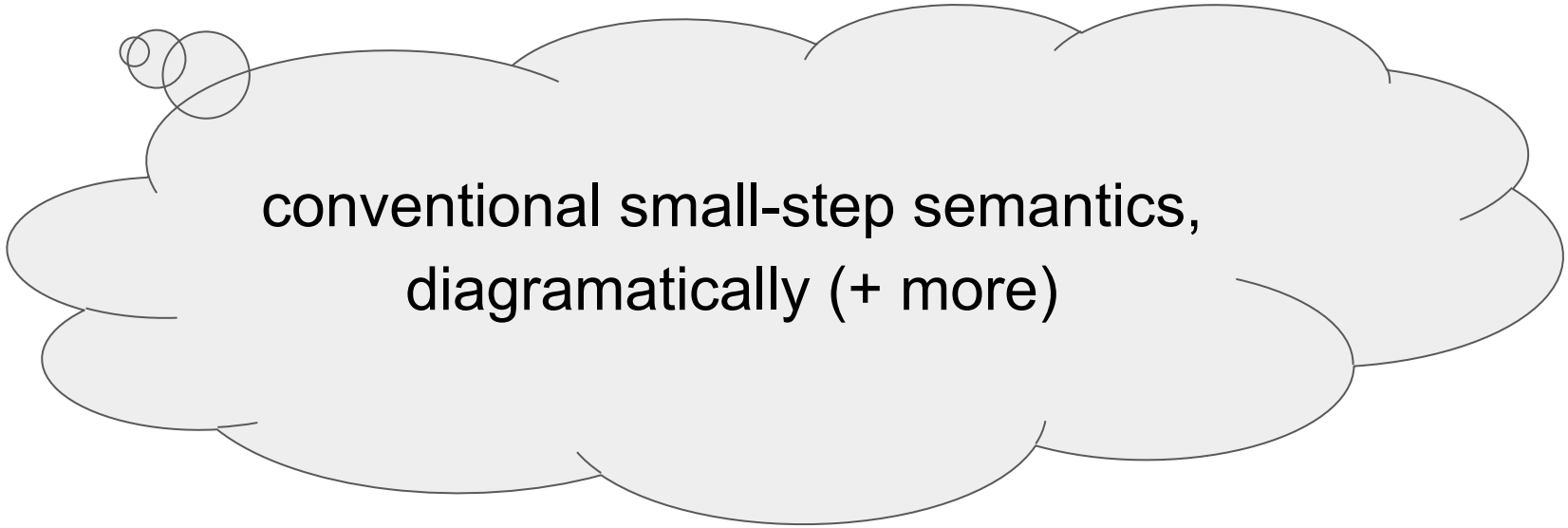
- compiler [Mackie ‘95]
- higher-order synthesis [Ghica ‘07]

Token-passing semantics *with* rewriting

Token-passing semantics *with* rewriting

execution models of functional programming

- call-by-name & call-by-value λ-calculus [Sinot ‘05]
- call-by-need & fully lazy λ-calculus [Sinot ‘06]
- call-by-need & call-by-value λ-calculus [−&Ghica ‘17]



Token-passing semantics *with* rewriting

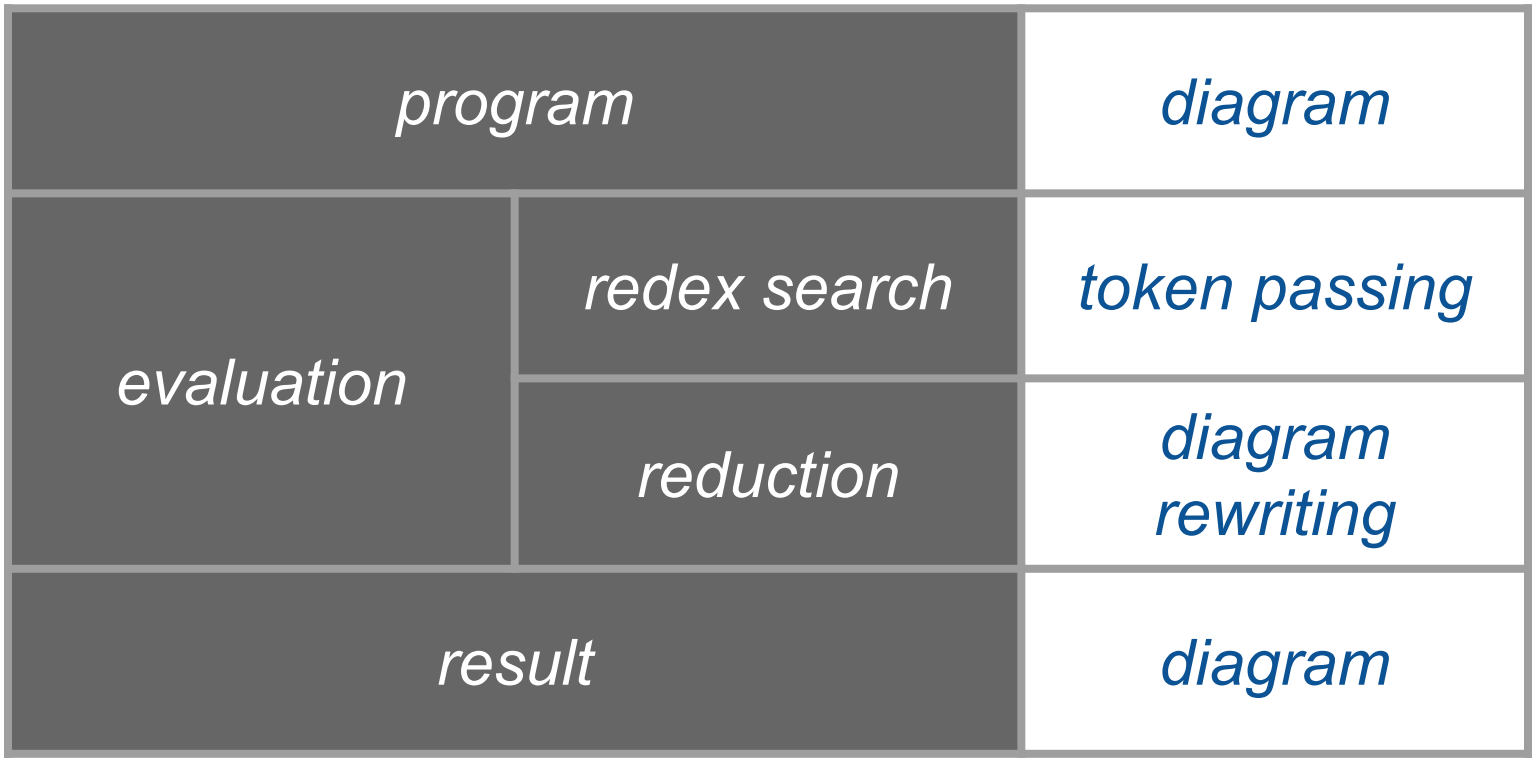
execution models of functional programming

- call-by-name & call-by-value λ-calculus [Sinot '05]
- call-by-need & fully lazy λ-calculus [Si
- call-by-need & call-by-value λ-calculus

inspired by virtual reduction [Danos&Regnier '93]

program		*diagram*
evaluation	*redex search*	*token passing*
	reduction	*diagram rewriting*

Token-passing semantics *with* rewriting

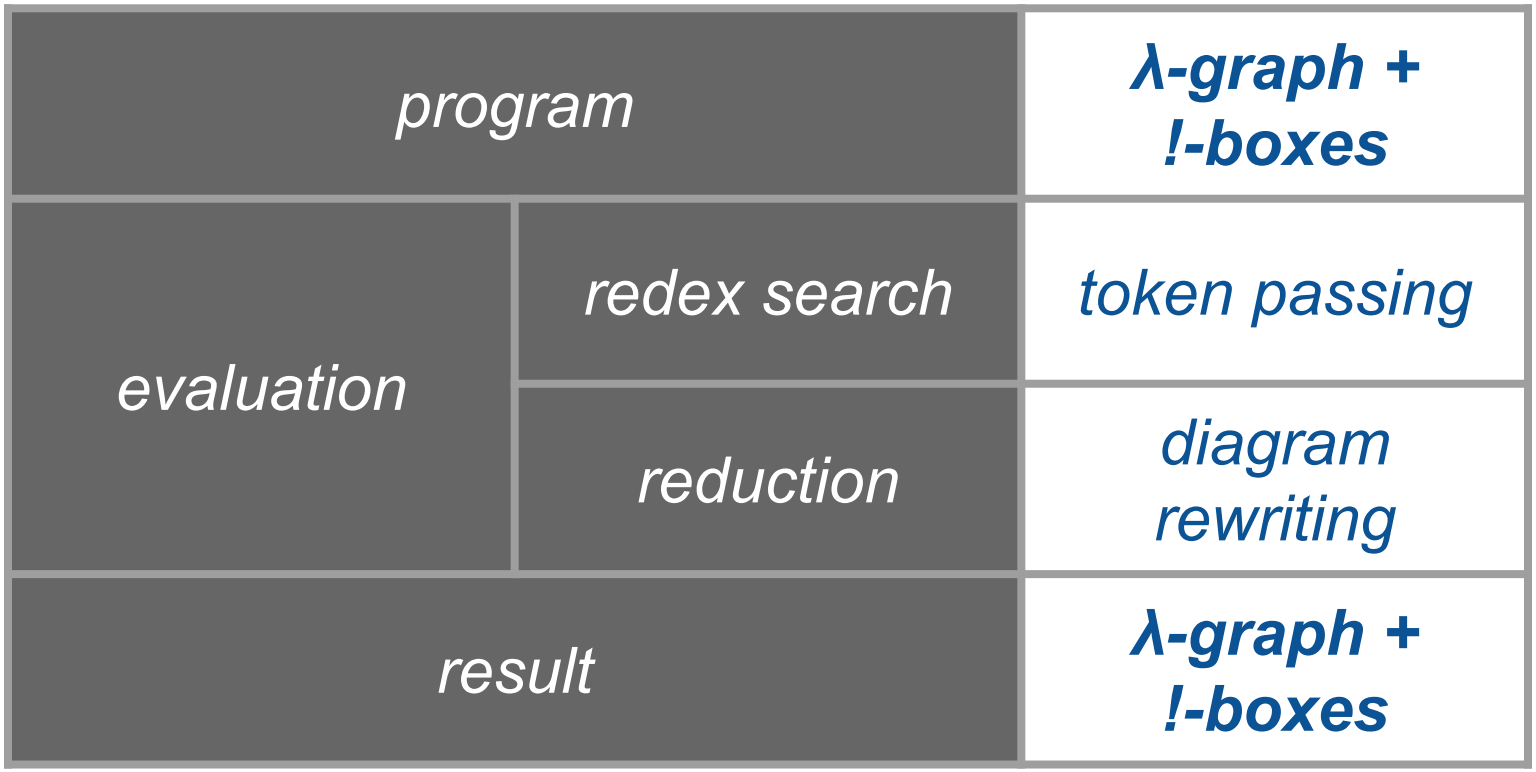


live demo:

Dynamic GoI Machine for call-by-value λ-calculus [KG17]

https://koko-m.github.io/GoI-Visualiser/

Token-passing semantics *with* rewriting

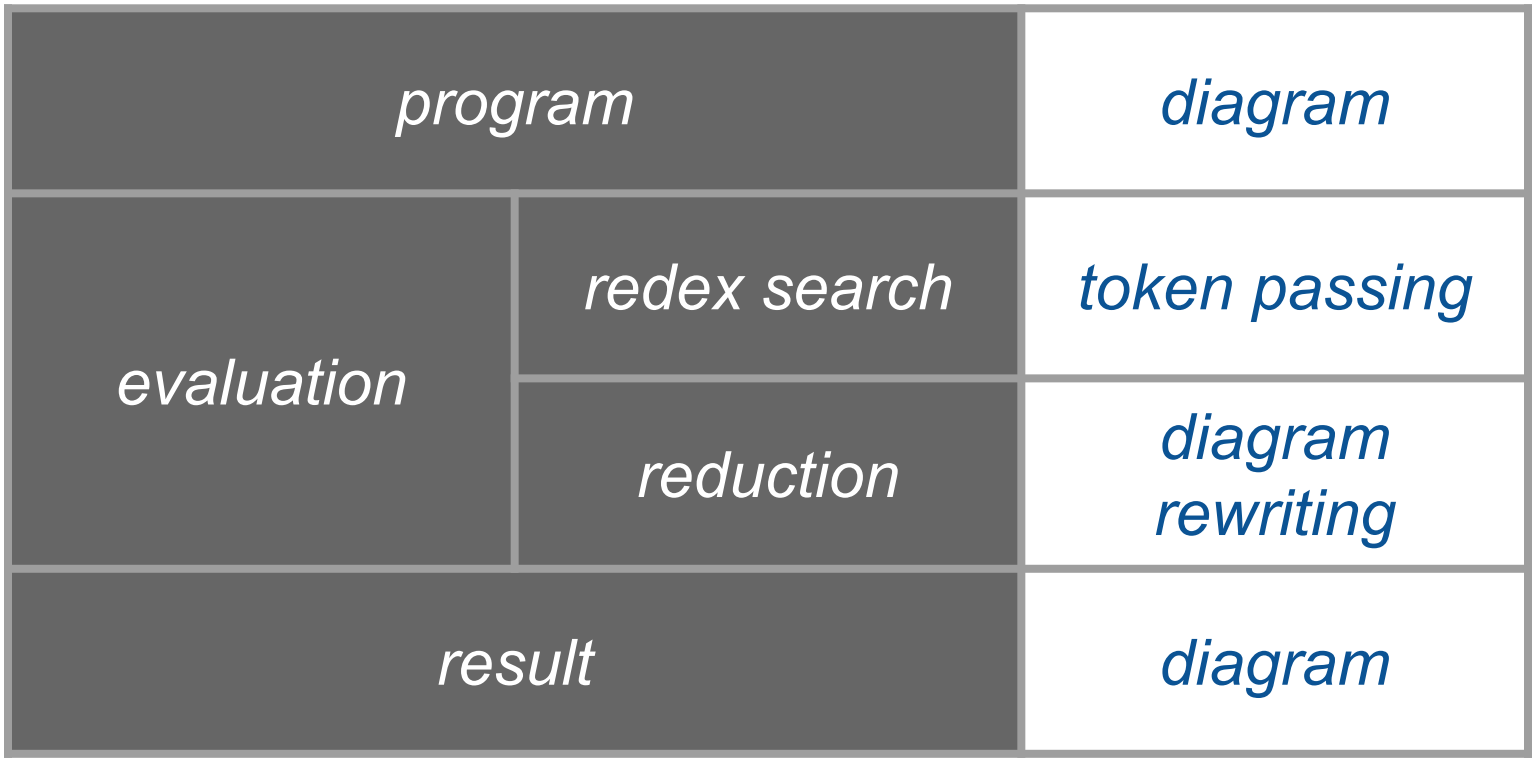


live demo:

Dynamic GoI Machine for call-by-value λ-calculus [KG17]

https://koko-m.github.io/GoI-Visualiser/

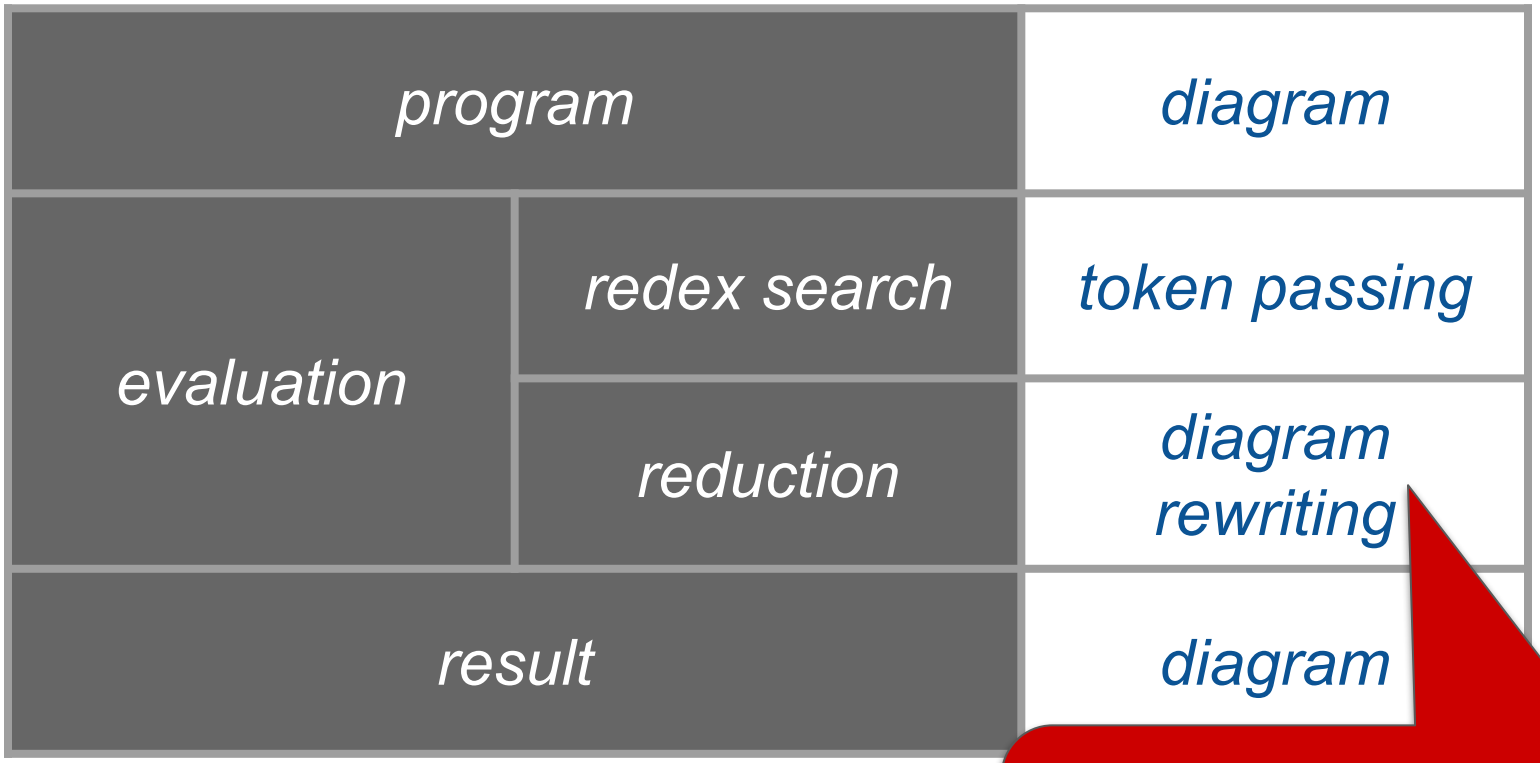
Token-passing semantics *with* rewriting



execution models of functional programming

- with robustness [S05] [S06]
- with time/space efficiency [MG17]

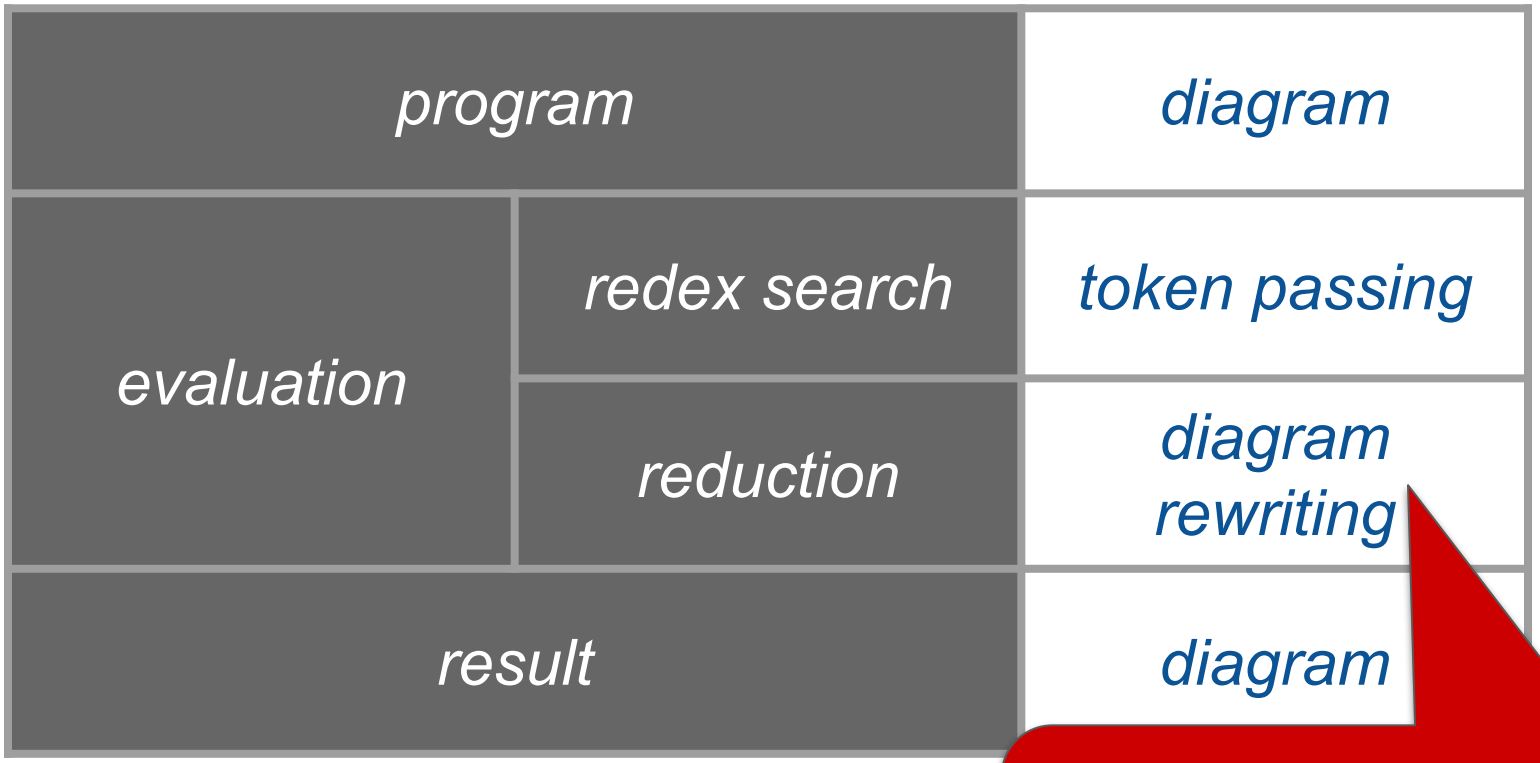
Token-passing semantics *with* rewriting



execution models of functional program

- with robustness [S05] [S06]
- with time/space efficiency [MG17]

Token-passing semantics *with* rewriting



execution models of functional program

- with robustness [S05] [S06]
- with time/space efficiency [MG17]

Token-passing semantics *with and without* rewriting

token-passing semantics *without* rewriting

… space efficiency

token-passing semantics *with* rewriting

… time efficiency

selective rewriting

non-trivial space/time balancing?

token-passing semantics *without* rewriting

… space efficiency

token-passing semantics *with* rewriting

… time efficiency

selective rewriting

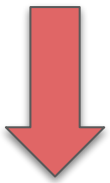
non-trivial space/time balancing?

token-passing semantics *without* rewriting

… result given by the token

token-passing semantics *with* rewriting

… result given by a diagram



programming with dual result

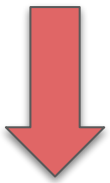
… given by the token *and* a diagram?

token-passing semantics *without* rewriting

… result given by the token

token-passing semantics *with* rewriting

… result given by a diagram



selective rewriting

programming with “computation graphs”

… result being value with computation graph

Programming with “computation graphs”

Programming with “computation graphs”

TensorFlow, Google’s machine-learning library

```
W = tf.Variable(...)
b = tf.Variable(...)
y = W * x_data + b

x_data = ...
y_data = ...
sess = tf.Session()
sess.run(init)
sess.run(train)

x_data = ...
sess = tf.Session()
sess.run(init)
y_initial_values =
sess.run(y)
```

construction

machine-learning model with parameters

manipulation

model training (imperative parameter update)

value extraction

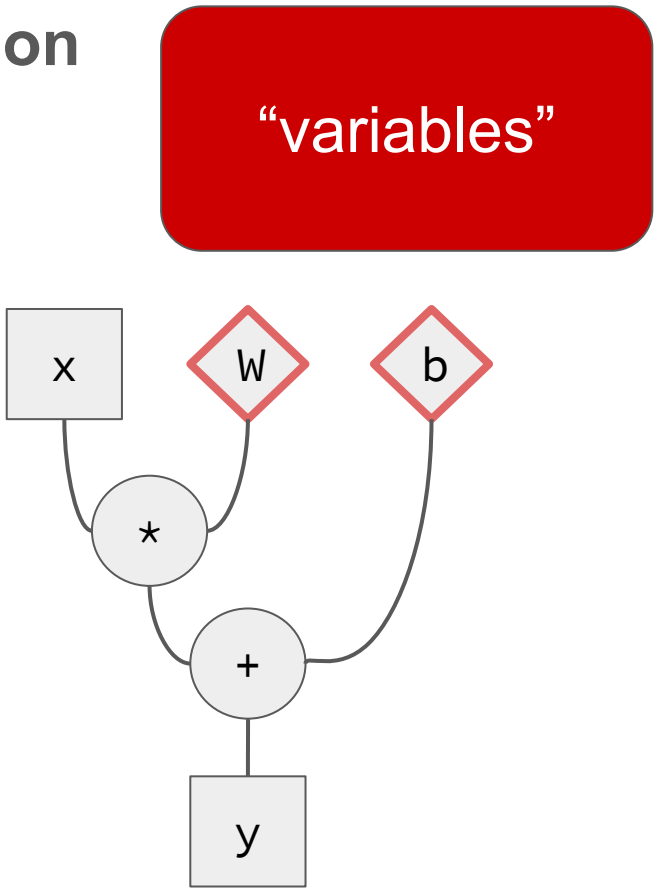
output prediction

Programming with “computation graphs”

TensorFlow, Google’s machine-learning library

```
W = tf.Variable(...)
b = tf.Variable(...)
y = W * x_data + b

x_data = ...
y_data = ...
sess = tf.Session()
sess.run(init)
sess.run(train)

x_data = ...
sess = tf.Session()
sess.run(init)
y_initial_values =
sess.run(y)
```

construction



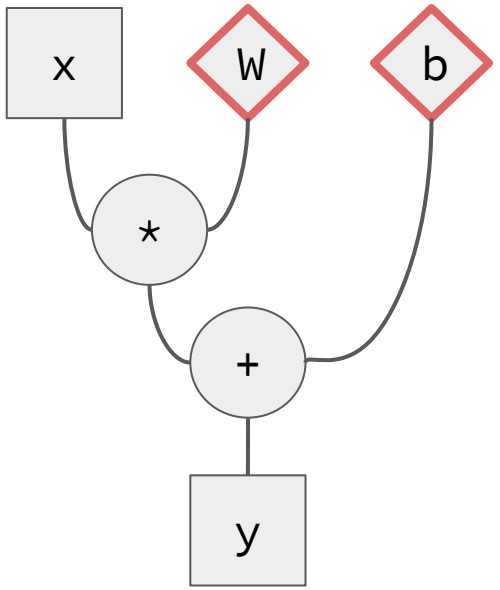
Programming with “computation graphs”

TensorFlow, Google’s machine-learning library

```
W = tf.Variable(...)
b = tf.Variable(...)
y = W * x_data + b

x_data = ...
y_data = ...
sess = tf.Session()
sess.run(init)
sess.run(train)

x_data = ...
sess = tf.Session()
sess.run(init)
y_initial_values =
sess.run(y)
```

construction

~~“variables”~~
“parameters”



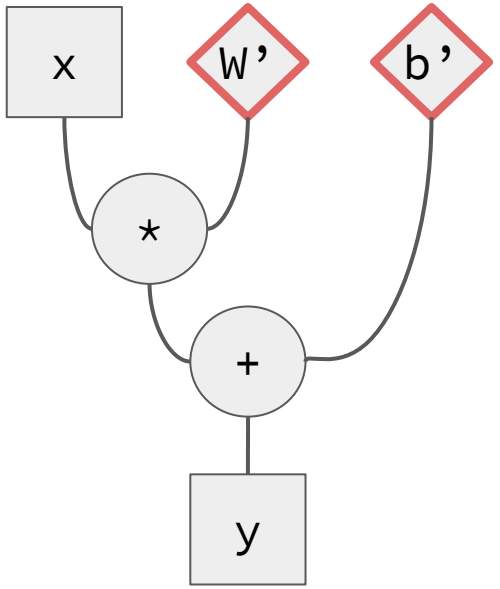
Programming with “computation graphs”

TensorFlow, Google’s machine-learning library

```
W = tf.Variable(...)
b = tf.Variable(...)
y = W * x_data + b

x_data = ...
y_data = ...
sess = tf.Session()
sess.run(init)
sess.run(train)

x_data = ...
sess = tf.Session()
sess.run(init)
y_initial_values =
sess.run(y)
```

manipulation

in-place (imperative) parameter update

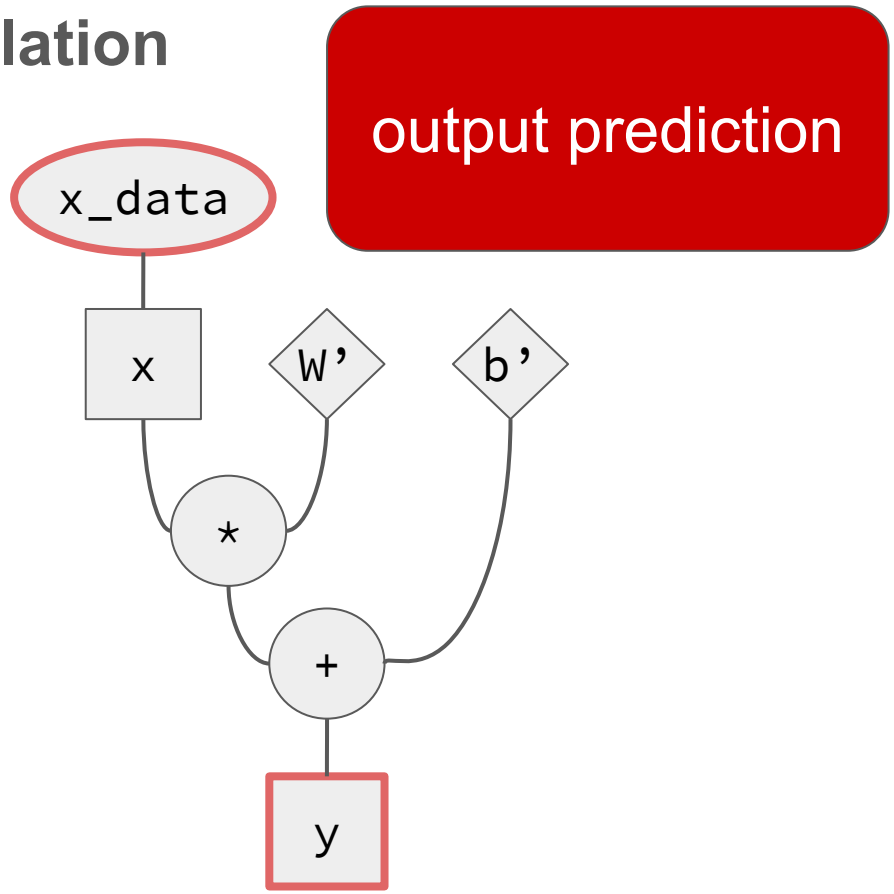


Programming with “computation graphs”

TensorFlow, Google’s machine-learning library

```
W = tf.Variable(...)
b = tf.Variable(...)
y = W * x_data + b

x_data = ...
y_data = ...
sess = tf.Session()
sess.run(init)
sess.run(train)

x_data = ...
sess = tf.Session()
sess.run(init)
y_initial_values =
sess.run(y)
```

manipulation



Programming with “computation graphs”

Self-Adjusting Computation [Acar ‘05]

(Incremental, an OCaml library)


```
let x = Inc.Var.create 1 in
let y = Inc.map2
 (Inc.Var.watch x)
 (Inc.Var.watch x) ~f:( + ) in
let z = Inc.Var.create 2 in
let w = Inc.map2
 (Inc.Var.watch y)
 (Inc.Var.watch z) ~f:( + ) in
let w_obs = Inc.observe w in
Inc.Var.set x 3;

Inc.stabilize ();

print_int
 (Inc.Observer.value_exn w_obs)
```

construction

acyclic dependency graph with “modifiables/cells”

manipulation

change propagation

value extraction

Programming with “computation graphs”

Self-Adjusting Computation [Acar ‘05]

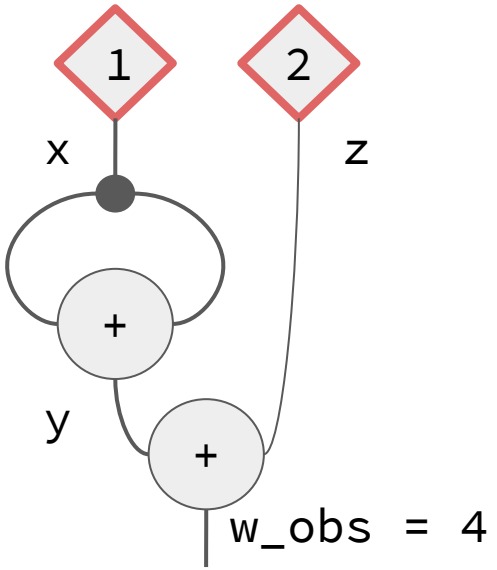
(Incremental, an OCaml library)

“spreadsheet”

```
let x = Inc.Var.create 1 in
let y = Inc.map2
 (Inc.Var.watch x)
 (Inc.Var.watch x) ~f:( + ) in
let z = Inc.Var.create 2 in
let w = Inc.map2
 (Inc.Var.watch y)
 (Inc.Var.watch z) ~f:( + ) in
let w_obs = Inc.observe w in
Inc.Var.set x 3;

Inc.stabilize ();

print_int
 (Inc.Observer.value_exn w_obs)
```

construction

“modifiables”
“cells”



Programming with “computation graphs”

Self-Adjusting Computation [Acar ‘05]

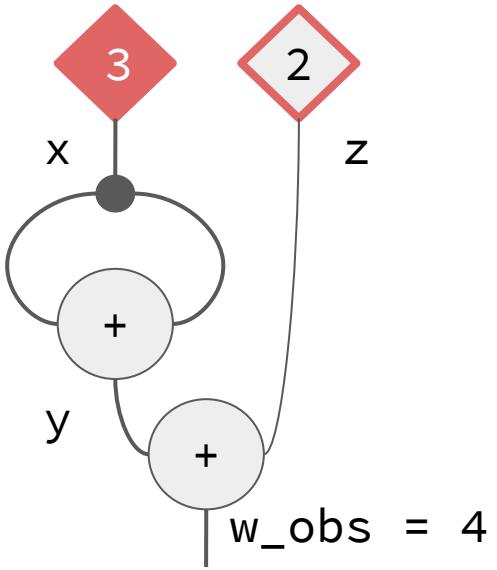
(Incremental, an OCaml library)

“spreadsheet”

```
let x = Inc.Var.create 1 in
let y = Inc.map2
 (Inc.Var.watch x)
 (Inc.Var.watch x) ~f:( + ) in
let z = Inc.Var.create 2 in
let w = Inc.map2
 (Inc.Var.watch y)
 (Inc.Var.watch z) ~f:( + ) in
let w_obs = Inc.observe w in
Inc.Var.set x 3;

Inc.stabilize ();

print_int
 (Inc.Observer.value_exn w_obs)
```

construction

change set



Programming with “computation graphs”

Self-Adjusting Computation [Acar ‘05]

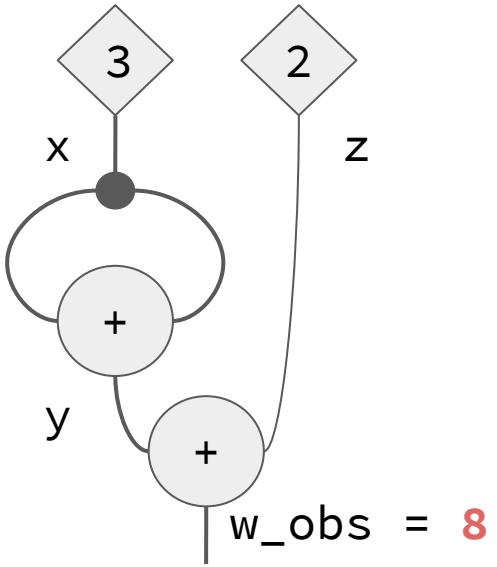
(Incremental, an OCaml library)

“spreadsheet”

```
let x = Inc.Var.create 1 in
let y = Inc.map2
 (Inc.Var.watch x)
 (Inc.Var.watch x) ~f:( + ) in
let z = Inc.Var.create 2 in
let w = Inc.map2
 (Inc.Var.watch y)
 (Inc.Var.watch z) ~f:( + ) in
let w_obs = Inc.observe w in
Inc.Var.set x 3;

Inc.stabilize ();

print_int
 (Inc.Observer.value_exn w_obs)
```

manipulation

change propagation



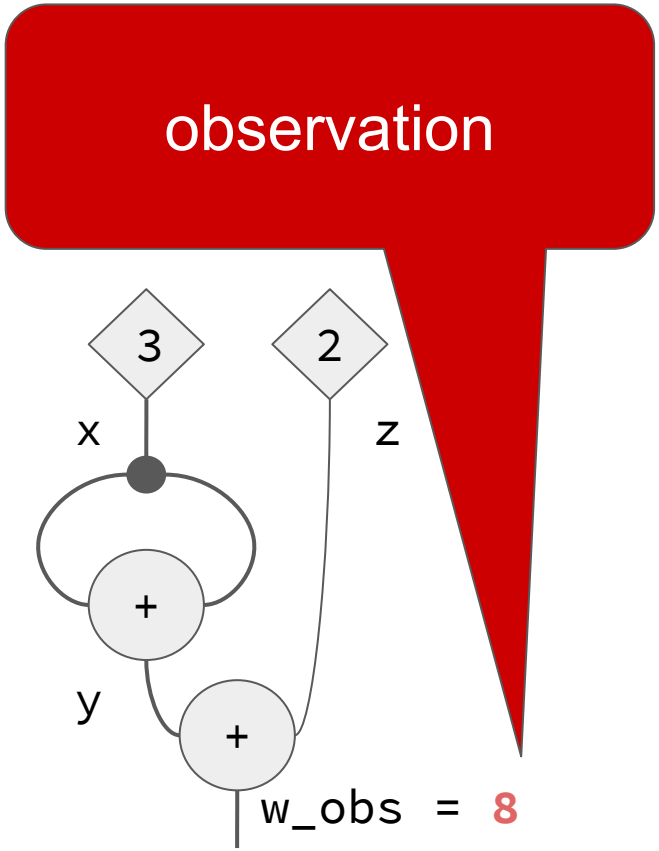
Programming with “computation graphs”

Self-Adjusting Computation [Acar ‘05]

(Incremental, an OCaml library)

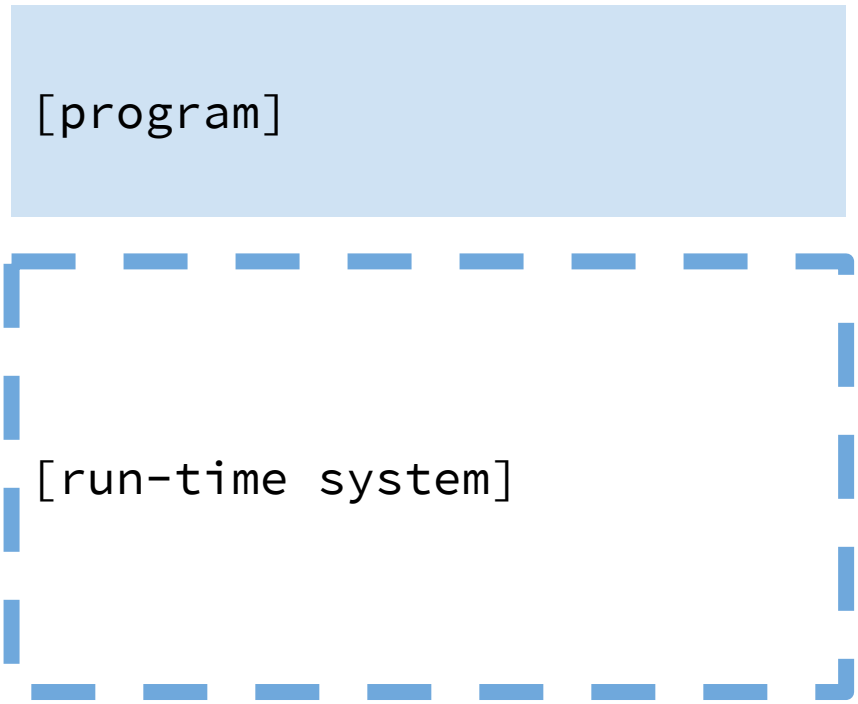

```
let x = Inc.Var.create 1 in
let y = Inc.map2
 (Inc.Var.watch x)
 (Inc.Var.watch x) ~f:( + ) in
let z = Inc.Var.create 2 in
let w = Inc.map2
 (Inc.Var.watch y)
 (Inc.Var.watch z) ~f:( + ) in
let w_obs = Inc.observe w in
Inc.Var.set x 3;

Inc.stabilize ();

print_int
 (Inc.Observer.value_exn w_obs)
```

value extraction



Programming with “computation graphs”

Probabilistic Programming



construction
stochastic model + observations

manipulation
stochastic inference

value extraction
posterior distribution

Programming with “computation graphs”

TensorFlow

imperative parameter update on machine-learning model

Self-Adjusting Computation

change propagation on acyclic dependency graph

Probabilistic Programming

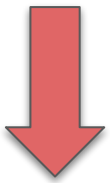
inference on stochastic model

token-passing semantics *without* rewriting

… result given by the token

token-passing semantics *with* rewriting

… result given by a diagram



selective rewriting

programming with “computation graphs”

… result being value with computation graph

Token-passing semantics *with & without* rewriting

program		*diagram*
evaluation	*redex search*	*token passing*
	reduction	*diagram rewriting*
computation graphs	*construction*	***selective*** *diagram rewriting*
	manipulation	*diagram rewriting*
	value extraction	*token data*

Programming with “computation graphs”

result	*value with computation graph*

- construction
- manipulation

Idealised TensorFlow [−,Cheung&Ghica ‘18]

functional parameter update on machine-learning model

Synchronous Self-Adjusting Computation

change propagation on cyclic dependency graph

Synchronous Self-Adjusting Computation

live demo: https://cwtsteven.github.io/GoI-SAC-Visualiser/

```
x = {1}
y = x + x
z = {2}
w = y + z
link x to 3;

_ = step ()
_ = step ()
w;
```

construction

cyclic dependency graph
with “modifiables/cells”

manipulation

cell-wise change propagation

multiple
independent tokens
without rewriting

Synchronous Self-Adjusting Computation

live demo: https://cwtsteven.github.io/GoI-SAC-Visualiser/

```
(* alternating signal *)

x = {true}
link x to ~x;

_ = step ()
_ = step ()
```

construction

cyclic dependency graph
with “modifiables/cells”

manipulation

cell-wise change propagation

multiple
independent tokens
without rewriting

token-passing semantics *without* rewriting

… result given by the token

token-passing semantics *with* rewriting

… result given by a diagram

selective rewriting

programming with "computation graphs"

… result being value with computation graph

given by the token and a diagram with "cells"

Token-passing semantics *with & without* rewriting

program		*diagram*
evaluation	*redex search*	*token passing*
	reduction	*diagram rewriting*
computation graphs	*construction*	***selective*** *diagram rewriting*
	manipulation	*diagram rewriting*
	value extraction	*token data*

Directions

- sit back and lay the foundation?
  - “cells”, special constants
    - shared but never duplicated
    - blocking rewrite

- more applications?
  - differentiating (higher-order) computation graphs
  - digesting meta-level stochastic inference