Diagrammatic execution models for functional programming

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

Steven W. T. Cheung
Dan R. Ghica
(University of Birmingham)
What happens when you run a functional program by hand?
Running a functional program by hand

(* computing the identity function in OCaml *)
(fun _ -> fun x -> fun x) ((fun y -> y) (fun z -> z))

-- computing the identity function in Haskell
(_ -> \x -> x) ((\y -> y) (\z -> z))

(\w. \x. x) ((\y. y) (\z. z))
Running a functional program by hand

(* computing the identity function in OCaml *)
(fun _ -> fun x -> fun x) ((fun y -> y) (fun z -> z))

(* => (fun _ -> fun x -> x) (fun z -> z) *)
(* => fun x -> x *)

(call-by-value evaluation)
Running a functional program by hand

-- computing the identity function in Haskell
\(_ \to \ x \to x\) ((\y \to y) (\z \to z))

-- \(_ \to \ x \to x\) ((\y \to y) (\z \to z))
-- \Rightarrow \ x \to x

call-by-need (lazy) evaluation
Running a functional program by hand

(* computing the identity function in OCaml *)
(fun _ -> fun x -> fun x) ((fun y -> y) (fun z -> z))

(* (fun _ -> fun x -> x) ((fun y -> y) (fun z -> z)) *)
(* => (fun _ -> fun x -> x) (fun z -> z) *)
(* => fun x -> x *)

-- computing the identity function in Haskell
(_ -> \x -> x) ((\y -> y) (\z -> z))

-- (\_ -> \x -> x) ((\y -> y) (\z -> z))
-- => \x -> x
Let’s run a functional program with a bit of formality...
Running a functional program, formally

by change of configuration

- program
- “focus”

(* computing the identity function in OCaml *)

(fun _ -> fun x -> fun x) ((fun y -> y) (fun z -> z))

call-by-value evaluation
Running a functional program, formally

by change of configuration

- program
- “focus”

(* computing the identity function in OCaml *)

(fun _ -> fun x -> fun x) ((fun y -> y) (fun z -> z))

call-by-value evaluation
Running a functional program, formally by change of configuration

- program
- “focus”

(* computing the identity function in OCaml *)

(fun _ -> fun x -> fun x) ((fun y -> y) (fun z -> z))

call-by-value evaluation
Running a functional program, formally

by change of configuration

- program
- “focus”

(* computing the identity function in OCaml *)

(fun _ -> fun x -> fun x) (fun z -> z)

call-by-value evaluation
Running a functional program, formally

by change of configuration

- program
- "focus"

(* computing the identity function in OCaml *)

(fun _ -> fun x -> fun x) (fun z -> z)
Running a functional program, formally by change of configuration

- program
- “focus”

(* computing the identity function in OCaml *)

```ocaml
fun x -> fun x
```

call-by-value evaluation
Running a functional program, formally

by change of configuration

- program
- “focus”

-- computing the identity function in Haskell
(\_ -> \x -> x) (((\y -> y) (\z -> z))

call-by-need (lazy) evaluation
Running a functional program, formally

by change of configuration

- program
- “focus”

-- computing the identity function in Haskell
\((\_ \rightarrow \_ \rightarrow x) ((\_ \rightarrow y) (\_ \rightarrow z))\)

call-by-need (lazy) evaluation
Running a functional program, formally

by change of **configuration**

- program
- “focus”

---

-- computing the identity function in Haskell
\[ x \rightarrow x \]
Running a functional program, formally

by change of configuration

- program
- “focus” determining evaluation strategy

(* computing the identity function in Ocaml *)

(fun _ -> fun x -> fun x) ((fun y -> y) (fun z -> z))

(* => (fun _ -> fun x -> x) (fun z -> z) *)
(* => fun x -> x *)

-- computing the identity function in Haskell

(_ -> x -> x) ((y -> y) (z -> z))

-- => \x -> x
prove program equivalence (validate compiler optimisations)

- “When are two programs the same?”
- result modelled by observable transitions & final configuration
Prove program equivalence (validate compiler optimisations)

- "When are two programs the same?"
- result modelled by observable transitions & final configuration

Analyse execution cost

- "How much time/space does it take to run a program?"
- time cost modelled by number and cost of transitions
- space cost modelled by size of configurations
Goodness of formality

prove program equivalence (validate compiler optimisations)

- “When are two programs the same?”
- result modelled by observable transitions & final configuration

analyse execution cost

- “How much time/space does it take to run a program?”
- time cost modelled by number and cost of transitions
- space cost modelled by size of configurations

guarantee “correctness” of implementation

- “Does a compiler work as intended?”
- implementation derived as abstract machine
Now we use *diagrams* to run a functional program with formality!
Conventional approaches

running a program by change of configuration

Diagrammatic approach

running a program by change of diagram configuration
Running a functional program with diagrams

by change of diagram configuration

- diagram representation of program
- “token” determining evaluation strategy

name-free
Running a functional program with diagrams

by change of *diagram configuration*

- diagram representation of program
- “token” determining evaluation strategy

[DEMO]

On-line visualiser for lambda-calculus

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

- call-by-name, call-by-need (lazy)
- call-by-value: left-to-right, right-to-left

name-free
Goodness of *diagrams*

- name-free
- environment included
Goodness of diagrams

visualise interesting properties of programs

● call-by-value vs. call-by-need (lazy)
● on-demand copying with intermediate sharing
● patterns in divergence
  ○ (\lambda x. x x) (\lambda x. x x)
  ○ (\lambda x. x x x) (\lambda x. x x x)
visualise interesting properties of programs

- call-by-value vs. call-by-need (lazy)
- on-demand copying with intermediate sharing
- patterns in divergence
  - \((\lambda x. x x) \ (\lambda x. x x)\)
  - \((\lambda x. x x x) \ (\lambda x. x x x)\)

answer (conventional) questions from new perspectives

- “When are two programs the same?”
- “How much time/space does it take to run a program?”
- “Does a compiler work as intended?”
More goodness of *diagrams*
More goodness of *diagrams*

support textual-and-visual programming

guide language design for unconventional/new programming paradigms (to be presented by Steven)
Towards textual-and-visual programming

“We’d like not just text or diagram, but both!”

(textual program)

\((\lambda w. \lambda x. x) ((\lambda y. y) (\lambda z. z))\)
Towards textual-and-visual programming

“We’d like not just text or diagram, but both!”

(\lambda w. \lambda x. x) ((\lambda y. y) (\lambda z. z))

<table>
<thead>
<tr>
<th>(grammar)</th>
<th>form / edit</th>
<th>(validity criteria)</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔</td>
<td>execute</td>
<td>✔</td>
</tr>
<tr>
<td>✔</td>
<td>debug</td>
<td>!</td>
</tr>
</tbody>
</table>
OCaml Visual Debugger

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

for a subset of OCaml

- arithmetic (int), comparison (bool)
- conditional (if), recursion (let rec)
- lists, pairs
- pattern matching

features

- interactive diagram view
- go forwards/backwards & pause/resume & jump steps
- breakpoint on diagram
- stats
OCaml Visual Debugger

visualise interesting properties of programs

- sorting algorithms comparison
  - bubble-sort vs. insert-sort
    https://www.youtube.com/watch?v=bZMSwo0zLio&t=130s
  - merge-sort vs. insert-sort
    https://www.youtube.com/watch?v=U1NI-mWeNe0
- tail-recursion vs. non tail-recursion
  https://www.youtube.com/watch?v=R4yCV5Ts1gk&t=14s
support textual-and-visual programming

- “We’d like not just text or diagram, but both!”
- We’ve got OCaml Visual Debugger
- … and want a text-and-diagram editor!

guide language design for unconventional/new programming paradigms (to be presented by Steven)
More goodness of *diagrams*

support textual-and-visual programming

- “We’d like not just text or diagram, but both!”
- We’ve got OCaml Visual Debugger
- … and want a text-and-diagram editor!

guide language design for unconventional/new programming paradigms (to be presented by Steven)