REPL-first Exploratory Programming

First conclusions and follow-up questions

L. Thomas van Binsbergen

Informatics Institute, University of Amsterdam
ltvanbinsbergen@acm.org

March 10, 2021

Evaluation approach

Approach

- Command-line interface for Funcons-beta
- Command-line interface, web-interface, and actor-oriented interface for eFLINT

Evaluation approach

Approach

- Command-line interface for Funcons-beta
- Command-line interface, web-interface, and actor-oriented interface for eFLINT

```
Linking dist/build/funcons-repl/funcons-repl ...
thomas@ltpro:~/repos/exploring-interpreters/funcons-tools-0.2.0.9$ dist/build/funcons-repl/funcons-repl
#1 > bind("x",1)
#2 > print(bound("x"))
1#2 > integer-add(1,2)
#2 > bind("y", alloc(values))
#3 > assign(bound("y"),bound("x"))
#4 > print(assigned(bound("y")))
1#4 >
```

Figure: Funcons-beta example involving output, binding and storing

Evaluation approach

Approach

- Command-line interface for Funcons-beta
- Command-line interface, web-interface, and actor-oriented interface for eFLINT

Questions

- Advantages and disadvantages of backtracking and/or/nor sharing?
- What functionality is needed of the exploring interpreter to support the various interfaces?
- Is the formal model of ONWARD2020 sufficient? (i.e. def. of languages and algorithm)

Invariants

- Retaining the genericity of the back-end
- Back-end retains canonical exploratory state
- Avoiding code-duplication in the implementation

Single-Trace, Single-Head (STSH) exploration

- Stack-like behaviour: destructive backtracking, no sharing
- Current node (dashed square) always top of stack

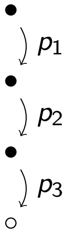


Figure: Trace of r_3

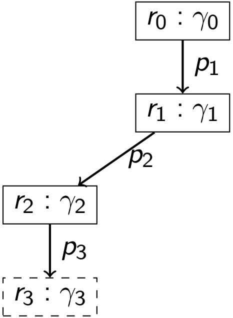


Figure: Execution graph after execution $p_1 \ldots p_3$

Single-Trace, Single-Head (STSH) exploration

- Stack-like behaviour: destructive backtracking, no sharing
- Current node (dashed square) always top of stack

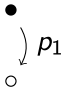


Figure: Trace of r_1

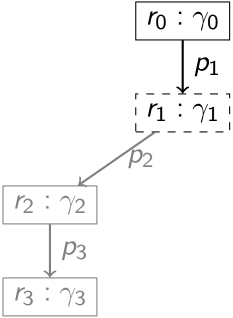


Figure: Execution graph after execution $p_1 \ldots p_3$ and reverting to r_1

Single-Trace, Single-Head (STSH) exploration

- Stack-like behaviour: destructive backtracking, no sharing
- Current node (dashed square) always top of stack

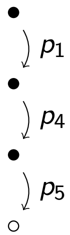


Figure: Trace of r_5

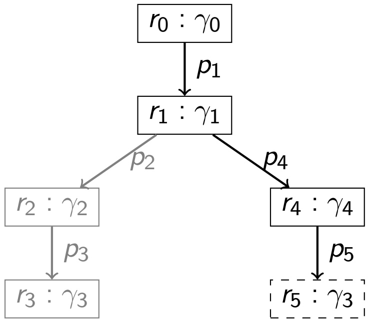


Figure: Execution graph after execution $p_1 \ldots p_3$ and reverting to r_1 and executing p_4, p_5

Single-Trace, Multi-Head (STMH) exploration

- Tree-traversal: non-destructive reverting, no sharing
- Multiple paths explored simultaneously

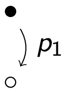


Figure: Trace of r_1

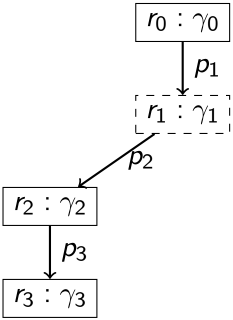


Figure: Execution graph after execution $p_1 \ldots p_3$ and reverting to r_1

Single-Trace, Multi-Head (STMH) exploration

- Tree-traversal: non-destructive reverting, no sharing
- Multiple paths explored simultaneously

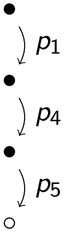


Figure: Trace of r_5

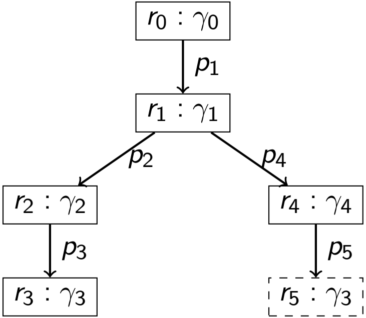


Figure: Execution graph after execution $p_1 \ldots p_3$ and reverting to r_1 and executing p_4, p_5

Multi-Trace, Multi-Head (MTMH) exploration

- Graph-traversal: non-destructive reverting, with sharing
- Multiple paths explored, multiple traces on current node

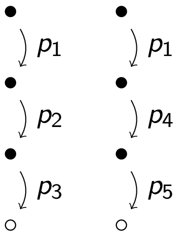


Figure: Traces of r_3

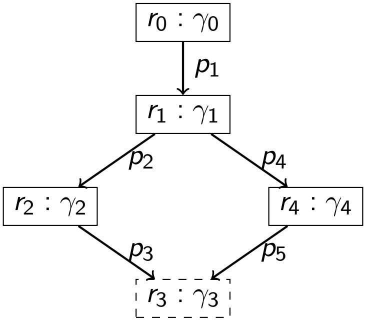


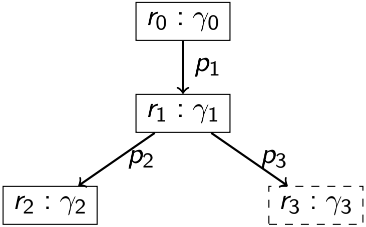
Figure: Execution graph after execution $p_1 \ldots p_3$ and reverting to r_1 and executing p_4, p_5

Questions

- Advantages and disadvantages of backtracking and/or/nor sharing?
- What functionality is needed of the exploring interpreter to support the various interfaces?
- Is the formal model of ONWARD2020 sufficient? (i.e. def. of languages and algorithm)

Discussion on destructive backtracking

Non-destructive reverting is needed for ‘true’ exploratory programming (i.e. Multi-Head)



Discussion on destructive backtracking

Non-destructive reverting is needed for ‘true’ exploratory programming (i.e. Multi-Head)

$$\begin{tikzcd} & \boxed{r_0 : \gamma_0} \arrow[d, "p_1"] & \\ & \boxed{r_1 : \gamma_1} \arrow[dl, "p_2"'] \arrow[dr, "p_3"] & \\ \boxed{r_2 : \gamma_2} & & r_3 : \gamma_3 \end{tikzcd}$$

However, certain applications can use destructive backtracking to save time and space, e.g.

- Time: batch testing many tests with a common (costly) prefix (e.g. interpretation in server mode)
- Space: simulations performed with eFLINT normative actors

Discussion on destructive backtracking

Non-destructive reverting is needed for ‘true’ exploratory programming (i.e. Multi-Head)

$$
\begin{tikzcd}
& \boxed{r_0 : \gamma_0} \arrow[d, "p_1"] & \\
& \boxed{r_1 : \gamma_1} \arrow[dl, "p_2"'] \arrow[dr, "p_3"] & \\
\boxed{r_2 : \gamma_2} & & r_3 : \gamma_3
\end{tikzcd}
$$

However, certain applications can use destructive backtracking to save time and space, e.g.

- Time: batch testing many tests with a common (costly) prefix (e.g. interpretation in server mode)
- Space: simulations performed with eFLINT normative actors

Decision can easily be based on *per application* or *per revert* basis

Potential advantages of sharing (1)

Detecting cycles and convergence. Is this useful in exploratory programming?

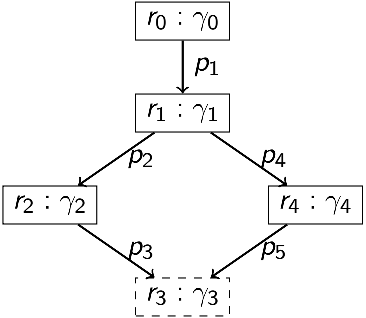


Figure: Convergence

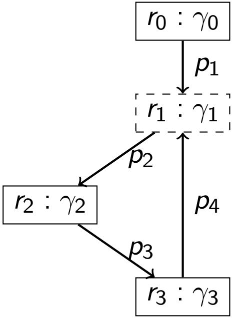


Figure: Cycle

Potential advantages of sharing (1)

Detecting cycles and convergence. Is this useful in exploratory programming?

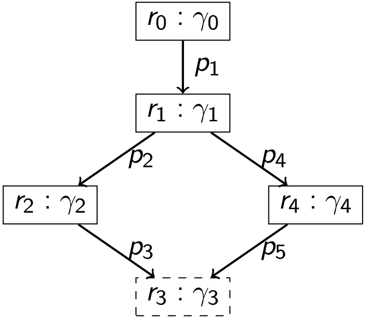


Figure: Convergence

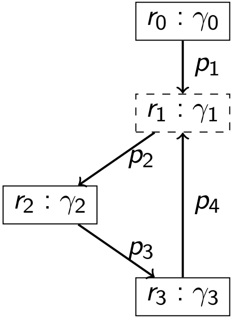


Figure: Cycle

What notion of equality to use to detect sharing? structural equality?

Potential advantages of sharing (2)

Detecting repeated computation before execution, e.g. avoiding $p_5 \equiv p_2$

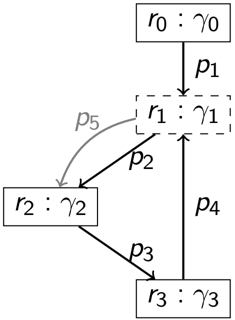


Figure: If $p_5 \equiv p_2$, then executing p_5 in r_1 can be skipped

Requires detecting equivalence to be effective, e.g. via normalisation

Disadvantages of sharing (1)

Ambiguity of revert, e.g. what is the trace of r_3? i.e. two 'histories' in r_3

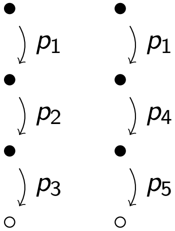
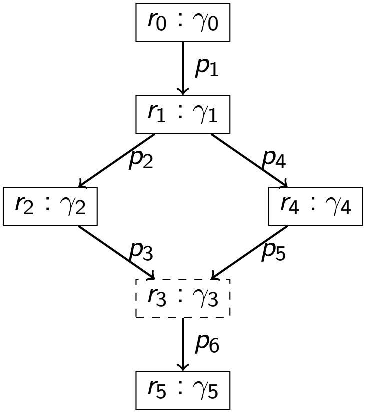


Figure: Traces of r_3.



Possible solutions: keep track of actions, order incoming edges, or clickable traces

Disadvantages of sharing (1)

Ambiguity of revert, e.g. what is the trace of r_3? i.e. two 'histories' in r_3

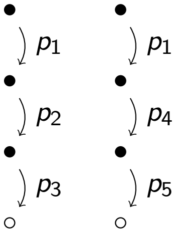
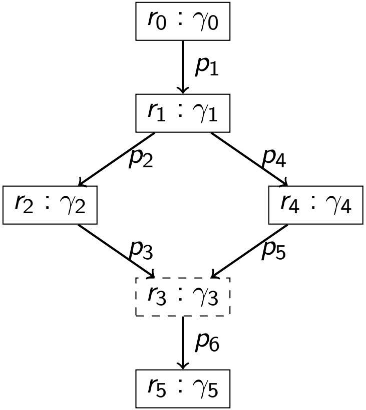


Figure: Traces of r_3.



Another complication: what edges/paths to remove when reverting to r_1?

Disadvantages of sharing (2)

- Multiples traces per node may not align with programmer's mental model
- Infinitely many traces (generated) when there is a cycle
- Possible ambiguity when reverting, i.e. when given reference or configuration
- Advantages may be marginal; this requires further, practical studies

Discussion on output – simulated I/O

In the formal model the definitional interpreter is pure:

$$\mathit{interpreter} : \mathit{program} \times \mathit{config} \rightarrow \mathit{config}$$

This then require the use of 'simulated' input and output (I/O) captured inside configurations

$$\boxed{[\,]} \xrightarrow{\mathit{print}(1)} \boxed{[1]} \xrightarrow{\mathit{print}(2)} \boxed{[1, 2]} \xrightarrow{\mathit{print}(3)} \boxed{[1, 2, 3]}$$

Figure: Example of simulated output

Problem: every printed value gives rise to a new 'execution phase' with no possibility to reach configurations of earlier phases through program execution (only through reverts)

Discussion on output – real I/O

Use an impure function instead (e.g. using Haskell's IO monad or arbitrary monad m):

$$interpreter : program \times config \to IO\ config$$
$$interpreter : program \times config \to m\ config$$

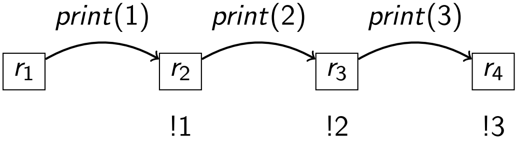


Figure: Real output, without sharing

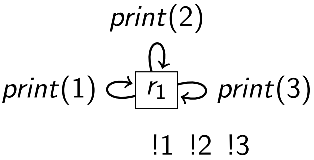


Figure: Real output, with sharing

With an arbitrary monad m: choose which effects to consider side-effects
Problem: monad (e.g. real input) determines soundness of the graph (e.g. input changes!)

Real I/O – Funcons-beta example

```
Linking dist/build/funcons-repl/funcons-repl ...
thomas@ltpro:~/repos/exploring-interpreters/funcons-tools-0.2.0.9$ dist/build/funcons-repl/funcons-repl
#1 > bind("x",1)
#2 > print(bound("x"))
1#2 > integer-add(1,2)
#2 > bind("y", alloc(values))
#3 > assign(bound("y"),bound("x"))
#4 > print(assigned(bound("y")))
1#4 >
```

Figure: Funcons-beta example involving output, binding and storing

```
thomas@ltpro:~/repos/exploring-interpreters/funcons-tools-0.2.0.9$ dist/build/funcons-repl/funcons-repl
#1 > bind("x",allocate-initialised-variable(values,read))

> 42
#2 > print(assigned(bound("x")))
42#2 >
```

Figure: Funcons-beta example involving input, output, binding and storing

Discussion on output – explicit, simulated output

Pure definitional interpreter with explicit output in its result
Label edges in the execution graph also with program output (enables refreshing)

$$interpreter : program \times config \to config \times output$$

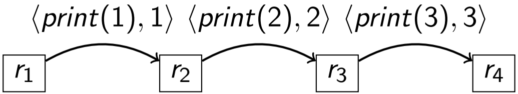


Figure: Explicit output, without sharing

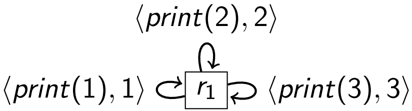


Figure: Explicit output, with sharing

How to display traces with output but no other effects?

Conclusions

Multiple variants of exploratory programming are possible, each with possible use-cases.

Conclusions

Multiple variants of exploratory programming are possible, each with possible use-cases.

Further studies are required to investigate the value of sharing.

Conclusions

Multiple variants of exploratory programming are possible, each with possible use-cases.

Further studies are required to investigate the value of sharing.

Several extensions/additions to formal model:

- References instead of configurations in nodes
- Extended definitional interpreters with output component
- Variants of **display**: last edge, path(s) from root to current, ...

Conclusions

Multiple variants of exploratory programming are possible, each with possible use-cases.

Further studies are required to investigate the value of sharing.

Several extensions/additions to formal model:

- References instead of configurations in nodes
- Extended definitional interpreters with output component
- Variants of **display**: last edge, path(s) from root to current, ...

The following combination works for all tested applications (Funcons-beta and eFLINT):

Conclusions

Multiple variants of exploratory programming are possible, each with possible use-cases.

Further studies are required to investigate the value of sharing.

Several extensions/additions to formal model:

- References instead of configurations in nodes
- Extended definitional interpreters with output component
- Variants of **display**: last edge, path(s) from root to current, ...

The following combination works for all tested applications (Funcons-beta and eFLINT):

- Sharing disabled

Conclusions

Multiple variants of exploratory programming are possible, each with possible use-cases.

Further studies are required to investigate the value of sharing.

Several extensions/additions to formal model:

- References instead of configurations in nodes
- Extended definitional interpreters with output component
- Variants of **display**: last edge, path(s) from root to current, ...

The following combination works for all tested applications (Funcons-beta and eFLINT):

- Sharing disabled
- Destructive backtracking / non-destructive revert on per application basis in eFLINT:
  - Destructive backtracking: batch testing, scenario web-interface, and normative actors
  - Non-destructive reverting: command-line REPL and exploratory web-interface

Conclusions

Multiple variants of exploratory programming are possible, each with possible use-cases.

Further studies are required to investigate the value of sharing.

Several extensions/additions to formal model:

- References instead of configurations in nodes
- Extended definitional interpreters with output component
- Variants of **display**: last edge, path(s) from root to current, ...

The following combination works for all tested applications (Funcons-beta and eFLINT):

- Sharing disabled
- Destructive backtracking / non-destructive revert on per application basis in eFLINT:
    - Destructive backtracking: batch testing, scenario web-interface, and normative actors
    - Non-destructive reverting: command-line REPL and exploratory web-interface
- Real output or simulated (explicit) output on per application basis in eFLINT:
    - Real output **and** simulated output (reproducability): command-line REPL
    - Simulated output: web-interfaces and normative actors

Open questions

- Are there applications to Multi-Trace Single Head (MTSH, not discussed here)?

Open questions

- Are there applications to Multi-Trace Single Head (MTSH, not discussed here)?
- Version control systems as a form of (STMH) exploratory programming.
  Can we use the formal model to describe this kind of exploratory programming?

Open questions

- Are there applications to Multi-Trace Single Head (MTSH, not discussed here)?
- Version control systems as a form of (STMH) exploratory programming.
  Can we use the formal model to describe this kind of exploratory programming?
- Is it practical to try detect and prevent repeated computations when sharing is enabled?

Future work

- *What generic interface components can we develop on top of the different variants?*

Future work

- *What generic interface components can we develop on top of the different variants?*
- *Is there a role for detecting convergence and cycles through sharing?*

Future work

- *What generic interface components can we develop on top of the different variants?*
- *Is there a role for detecting convergence and cycles through sharing?*
- *How to display (parts of) the execution graph (such as configurations/nodes, programs/edges, traces, etc.)*?

Investigate the human-computer interaction aspect of REPL-first exploratory programming, starting with Single-Trace, Multi-Head (STMH)

REPL-first Exploratory Programming

First conclusions and follow-up questions

L. Thomas van Binsbergen

Informatics Institute, University of Amsterdam
ltvanbinsbergen@acm.org

March 10, 2021