A generic back-end for exploratory programming

Damian Frölich$^{1,2}$ and Thomas van Binsbergen$^1$

$^1$University of Amsterdam
$^2$VU Amsterdam

February 2021
Programming forms

- Edit → Compile → Run
  - Slow interactivity
  - Shallow interactivity

https://xkcd.com/303/
REPL

- Incremental programming
- Immediate feedback
- Provides some form of exploratory programming
Notebook

- Alternative interface
- Combines with literate programming
Inconsistent interfaces

jshell> int x;
x ==> 0
jshell>
    class A {
        public void run() {
            x++;
        }
    }

| created class A
jshell> A a = new A();
a ==> A@5ce65a89
jshell> a.run()
jshell> x
x ==> 1
Extension of a language

- (some) Interfaces require an extension on the original language
  - Not always documented
  - Independent of original language
Principled approach

- Definitional interpreter
- Difference between base and extension language
- Generic interfaces
- Exploratory programming

```plaintext
Config eval((Phrase)`<Expression e> ;`, Config c) = catchExceptions(collectBindings(setOutput(createBinding(eval(c, e)))));

Config eval((Phrase)`<Statement s>`', Config c) = catchExceptions(collectBindings(setOutput(exec(s, c))));

Config eval((Phrase)`<ClassDecl cd>`', Config c) = catchExceptions(collectBindings(declareClass(cd, c)));

Config eval((Phrase)`<VarDecl vd>`', Config c) = catchExceptions(collectBindings(declareVariables(vd, c)));

Config eval((Phrase)`<MethodDecl md>`', Config c) = catchExceptions(collectBindings(declareGlobalMethod(md, c)));

Config eval((Phrase)`<Phrase p1> <Phrase p2>`', Config c) = eval(p2, eval(p1, c));
```

1 Binsbergen et al. 2020
Exploratory programming

- Exploring interpreter
  - Current configuration
  - Execution graph
  - Operations
    - Display
    - Execute
    - Revert

```
empty
[(x, 2)]
x = 2
```

```
empty
\[[(x, \ 2)]\]
x = 2
```
Exploratory programming

- Exploring interpreter
  - Current configuration
  - Execution graph
  - Operations
    - Display
    - Execute
    - Revert

- Execution graph behaviour
  - Stack
  - Tree
  - Graph
  - Graph-structured Stack (GSS)

```
x = 2
[(x, 2)]
```
Behaviour examples

Tree behaviour

Graph behaviour
Definition
A language $L$ is a structure $\langle P, \Gamma, \gamma^0, I \rangle$ with:

- $P$ a set of programs,
- $\Gamma$ a set of configurations,
- $\gamma^0 \in \Gamma$ an initial configuration and
- $I$ a definitional interpreter assigning to each program $p \in P$ a function $I_p : \Gamma \rightarrow \Gamma$. 

```haskell
whileInterpreter :: Command -> Config -> Config
data Config = Config { cfgStore :: Store, cfgOutput :: Output }  
  type Store = Map String Literal  
  type Output = [String]  
  initialConfig = Config { cfgStore = empty, cfgOutput = [] } 
```
Definition
A language $L$ is a structure $\langle P, \Gamma, \gamma^0, I \rangle$ with:

- $P$ a set of programs,
- $\Gamma$ a set of configurations,
- $\gamma^0 \in \Gamma$ an initial configuration and
- $I$ a definitional interpreter assigning to each program $p \in P$ a function $I_p : \Gamma \rightarrow \Gamma$.

```
whileInterpreter :: Command -> Config -> Config

data Config = Config { cfgStore :: Store, cfgOutput :: Output }

type Store = Map String Literal

type Output = [String]

initialConfig = Config { cfgStore = empty, cfgOutput = [] }
```
Definition
A language $L = \langle P, \Gamma, \gamma^0, I \rangle$ is *sequential* if there is an operator $\otimes$ such that for every $p_1, p_2 \in P$ and $\gamma \in \Gamma$ it holds that $p_1 \otimes p_2 \in P$ and that $I_{p_1 \otimes p_2}(\gamma) = (I_{p_2} \circ I_{p_1})(\gamma)$.

While is sequential
\[
\text{whileInterpreter}(\text{Seq } p_1 p_2) \gamma = \text{whileInterpreter}(p_2 . \text{whileInterpreter}(p_1)) \gamma
\]
Definition
A language $L = \langle P, \Gamma, \gamma^0, I \rangle$ is sequential if there is an operator $\otimes$ such that for every $p_1, p_2 \in P$ and $\gamma \in \Gamma$ it holds that $p_1 \otimes p_2 \in P$ and that $I_{p_1 \otimes p_2}(\gamma) = (I_{p_2} \circ I_{p_1})(\gamma)$.

While is sequential

```
whileInterpreter (Seq p_1 p_2) gamma ==
  (whileInterpreter p_2 . whileInterpreter p_1) gamma
```
Contribution

Provide a generic exploring interpreter allowing experimentation with the different execution graph behaviours for different type of languages and interfaces.
Exploring interpreter

data Explorer programs configs = Explorer
{ defInterp :: programs -> configs -> configs
, config :: configs
, execEnv :: Gr _ programs
}
type Ref = Int

data Explorer programs configs = Explorer
{
  ... 
  , execEnv :: Gr Ref programs 
  , currRef :: Ref 
  , cmap :: Map Ref configs 
}
data Explorer programs configs = Explorer
{ ... 
  , sharing :: Bool
  , backtracking :: Bool
}

<table>
<thead>
<tr>
<th>No backtracking</th>
<th>Backtracking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sharing</td>
<td>Graph</td>
</tr>
<tr>
<td></td>
<td>GSS</td>
</tr>
<tr>
<td>No sharing</td>
<td>Tree</td>
</tr>
<tr>
<td></td>
<td>Stack</td>
</tr>
</tbody>
</table>
data Explorer programs configs = Explorer
{ defInterp :: programs -> configs -> configs
, config :: configs
, execEnv :: Gr Ref programs
, currRef :: Ref
, cmap :: Map Ref configs
, sharing :: Bool
, backtracking :: Bool
}
mkExplorerStack :: (a -> b -> b) -> b -> Explorer a b
mkExplorerStack = mkExplorer False True

mkExplorerTree :: (a -> b -> b) -> b -> Explorer a b
mkExplorerTree = mkExplorer False False

mkExplorerGraph :: (a -> b -> b) -> b -> Explorer a b
mkExplorerGraph = mkExplorer True False

mkExplorerGSS :: (a -> b -> b) -> b -> Explorer a b
mkExplorerGSS = mkExplorer True True
Operations

revert :: Ref -> Explorer p c -> Maybe (Explorer p c)
execute :: (Eq c, Eq p) => p -> Explorer p c -> Explorer p c
Unpacking

Sequence of programs results in one transition

```
Seq(x = 0, print(x))
```

```
x = 0
print(x)
```
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b

executeAll :: (Eq c, Eq p) => [p] -> Explorer p c -> Explorer p c
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b

executeAll :: (Eq c, Eq p) => [p] -> Explorer p c -> Explorer p c
executeAll = flip (foldr execute)
Future work

Side effects?

▶ execute :: Monad m => p -> Explorer p c -> m (Explorer p c)

Debugging

Interface exploration

▶ Different type of interfaces
▶ Integration with different exploration behaviours
Evaluation with eFLINT

frames

scenario

Box
response

OK
output

Back value one

Buyer

Seller

Main offer

receive_offer(Asset(@1,50))

accept_offer(Asset(@1,50))

initialised

duty-to-pay(Seller, 50)

duty-to-deliver(Buyer, @1)

violation by Buyer: duty-to-pay(Seller, 50)

suspend-delivery(Buyer, @1)

duty terminated: duty-to-deliver(Buyer, @1)
A generic back-end for exploratory programming

The implementation opens up exploration of the exploring interpreter in a language independent manner

Damian Frolich
University of Amsterdam and VU Amsterdam
d.frolich@uva.nl

Thomas van Binsbergen
University of Amsterdam
tvanbinsbergen@acm.org