A Generic Back-End for Exploratory Programming

Damian Frolich[1,2] and L. Thomas van Binsbergen[3][0000−0001−8113−2221]

[1] Department of Computer Science, Vrije Universiteit, Amsterdam, The Netherlands
[2] Informatics Institute, University of Amsterdam, The Netherlands
d.frolich@uva.nl
[3] Informatics Institute, University of Amsterdam, The Netherlands
ltvanbinsbergen@acm.org

Abstract. Exploratory programming is a form of incremental program development in which the programmer can try and compare definitions, receives immediate feedback and can simultaneously experiment with the language, the program and input data. Read-Eval-Print-Loop interpreters (REPLs) and computational notebooks are popular tools for exploratory programming. However, their usability, capabilities and user-friendliness are strongly dependent on the underlying interpreter and, in particular, on the ad hoc engineering required to ready the underlying interpreter for incremental program development. To break this dependency, this paper adopts a principled approach and implements a so-called exploring interpreter as a back-end to support various development environments for exploratory programming.
This paper contributes by presenting a generic Haskell implementation of the exploring interpreter – applicable to a large class of software languages – and demonstrates its usage to develop a variety of interfaces with a shared back-end, including command-line REPLs, computational notebooks and servers with reactive APIs. The design of the back-end is evaluated by defining a variety of interfaces for existing languages, including eFLINT, a domain-specific language for normative reasoning, and Funcons-beta, the language developed by the PLanCompS project to enable component-based operational semantics.

Keywords: Interpreters · Development environments · Operational semantics · Read-Eval-Print · Definitional interpreters.

1 Introduction

Read-Eval-Print-Loop interpreters (REPLs) provide an alternative form of programming to the traditional compile-edit-run cycle. Popular examples of REPLs include JShell for Java, IPython for Python, PsySH for PHP and GHCi for Haskell, which are either part of the language's distribution (JShell and GHCi) or provide additional features on top of the REPL of the distribution (IPython and PsySH). REPLs enable an incremental form of programming in which a program is developed as a sequence of smaller programs executed one-by-one with

```
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
```

Fig. 1. JShell interaction.

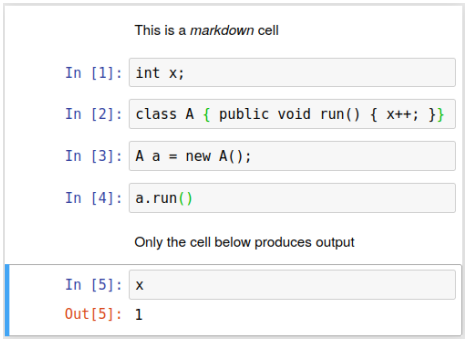

Fig. 2. IJava interaction.

Fig. 3. Example interactions in JShell and IJava.

immediate feedback after every (intermediate) program. This feedback typically includes the value computed by the program (in case of an expression) and a summary on the (side-)effects of the program, enabling the programmer to update their mental model of the REPLs underlying state. An example interaction with JShell is shown in Figure 1.

This quicker form of interaction, compared to the compile-edit-run cycle, makes REPLs more suitable for quickly testing library functions, retrieving (type) information on available bindings, experimenting with definitions, debugging, and analysing data. However, data analysts and other domain-experts, not necessarily skilled in software engineering, prefer to use computational notebooks for these tasks [32, 41]. Computational notebooks are documents consisting of a sequence of three types of cells: code cells, output cells and prose (or documentation) cells. Popular examples are Mathematica [13] and the notebooks built using the Jupyter platform [18]. Code cells are executed one-by-one, with output displayed in output cells, thereby supporting the same kind of incremental program development as REPLs. This is reflected in the design of the Jupyter platform, wherein Python notebooks use the IPython REPL internally [18]. An example of a Jupyter IJava notebook (based on JShell) is given in Figure 2.

REPLs and (Jupyter) notebooks require significant engineering, especially for languages, such as Java and to a lesser extent Haskell, that do not naturally support incremental program development. For example, the code fragments in Figure 3 can be recognised as Java code but they do not form a valid Java program individually nor as a sequence. JShell can be seen as implementing an extension of Java rather than Java itself. However, the precise details of this extension – its syntax and semantics – are not clearly specified and are not part of the Java documentation. Moreover, as Figure 3 demonstrates, JShell and IJava are not consistent in how they present output. In the example, JShell produces detailed information about the effects of most code fragments whereas IJava

only produces output for the last code fragment, revealing a difference between both tools in how they treat computed values and (side-)effects which, one could argue, are matters of language semantics rather than tool implementation.

In previous work [8], a principled approach is proposed for implementing REPLs, and other interfaces for incremental programming, using language engineering techniques to explicitly define language extensions, thereby clarifying the difference between the base language and the language implemented by the REPL. The approach makes it possible to develop generic interfaces which under the hood use a definitional interpreter[4] to execute programs. The approach further suggest the use of a so-called exploring interpreter on top of a definitional interpreter for *exploratory programming*. Exploratory programming is an open-ended form of incremental programming in which both the goal and the path towards the goal are discovered as part of the process [45, 3, 36]. The programmer discovers these through interactions with the underlying interpreter by testing definitions, evaluating expressions, analysing intermediate results and using backtracking to undo work and explore alternative directions.

1.1 Contributions

This paper contributes by presenting and discussing a generic implementation of the exploring interpreter algorithm of [8] in Haskell. The implementation is generic in the sense that it can be applied to large class of languages, including all languages that can have their semantics expressed by a transition function, for example in a transition system in the style of Plotkin [34].

Potential applications of the implemented algorithm are manifold. The genericity of the algorithm makes it possible to implement and experiment with features that benefit exploratory programming in a language-independent fashion. These features can then be used in a variety of interfaces and can be reused across languages. In other words, the exploring interpreter adds a level of indirection that makes it possible to deliver multiple programming interfaces for the same language by reusing the back-end and to deliver generic programming interfaces that can be reused across languages. Concretely, this paper:

- Presents a generic implementation of the exploring interpreter algorithm of [8] in Haskell and discusses the key design choices of the implementation
- Demonstrates the ability to reuse the algorithm as a back-end for various programming interfaces for exploratory programming and performs a qualitative evaluation on the implementation
- Applies the generic back-end to Funcons-beta [7] and eFLINT [6]. This effort made a significant, positive impact on the usability and applicability of these languages, demonstrating the practicality of the principled approach of [8]

[4] A definitional interpreter for a language is an interpreter that simultaneously implements and defines the language's operational semantics, often defined in a meta-language or language workbench in the context of domain-specific languages.

This paper is organised as follows. Section 2 and 3 describe background and related work. Section 4 presents an initial implementation of the exploring interpreter algorithm. Section 5 applies the algorithm to the Funcons-beta and eFLINT languages, demonstrating several types of front-ends for exploratory programming. To support these various types of front-ends, the initial implementation is extended in several ways in Section 5 as part of a qualitative evaluation. Section 6 concludes.

2 Background

This section introduces the methodology and related concepts put forward in [8]. In the proposed methodology, the first step towards developing a REPL for a language is to extend that language to a variant which is in the class of *sequential languages* – the class of languages that naturally support incremental program development. The class of sequential languages is defined in [8] as follows:

Definition 1. *A language L is a structure $\langle P, \Gamma, \gamma^0, I\rangle$ with P a set of programs, Γ 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 $I_p : \Gamma \to \Gamma$.*

Definition 2. *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 $I_{p_1 \otimes p_2}(\gamma) = (I_{p_2} \circ I_{p_1})(\gamma)$.*

The chosen definition of languages captures all software language that can have their semantics expressed as a deterministic transition function and includes real-world, large-scale, deterministic programming languages – as demonstrated by the body of literature on big-step, small-step and natural semantics [34, 1, 26, 15, 24] – and does not exclude languages with non-deterministic aspects when these aspects can be captured algebraically [48]. Configurations capture all information necessary to determine the behaviour of a program. A definitional interpreter is described as assigning to each program an *effect* – a function over configurations. A sequential language is a language in which every sequence of programs is a valid program that has the same effect as the composition of the effects of the individual programs in the sequence.

As an example, consider a simple imperative language such as While [1, 5]. In [5], a transition system is defined to capture the semantics of While commands. A configuration in this system contains a sequence of output values and a store to keep track of variable assignments. The system can be used to give a definitional interpreter for While, as required by Definition 1, for which it is possible to prove that $I_{C_1 \,;\, C_2}(\gamma) = (I_{C_2} \;\mathbf{;}\; I_{C_1})(\gamma)$, i.e. to prove that While is a sequential language according to Definition 2 by choosing $\mathbf{;}$ for $\otimes$.

The central idea of the approach is that an interpreter for a sequential language can be used, without (further) modification, by the back-end of a REPL, as well as by other interfaces for incremental programming. In other words, a REPL is considered to be just one type of interface for programming in the style

that is characteristic of REPLs. The precise behaviour of a programming interface is clarified by separating the task of building the interface into language engineering – producing a sequential variant of the base language and an interpreter – and the engineering required to link interface actions to the interpreter and to visualise the effects of programs.

The methodology further proposes the use of a so-called *exploring interpreter* to support exploratory programming. An exploring interpreter is a bookkeeping device on top of a definitional interpreter keeping track of executed programs and visited configurations. The **execute** action of an exploring interpreter for a language executes a program by applying the definitional interpreter for the language while keeping track of the encountered configurations and executed programs in an execution graph, reflecting the entire history of the current interactive session. The execution graph has configurations as nodes and edges between nodes are labelled with programs such that an edge between s and t labelled p indicates that executing p in the context of s yields t, i.e. $I_p(s) = t$. The **revert** action makes it possible to choose any (previously visited) configuration as providing the execution context for the next program, thereby enabling exploratory programming. If the language to which the generic algorithm is applied is a sequential language, then the execution graph of the resulting exploring interpreter is closed under transitivity. This property guarantees the soundness of a variety of operations on the graph.

3 Related work

Definitional interpreters of the kind captured by Definition 1 can be produced in a language workbench [11] such as Spoofax [16] or the $\mathbb{K}$ framework [19], a meta-language such as Rascal [17], a suitable general-purpose language such as Haskell [30, 23, 14], or can be generated from a formal definition of the operational semantics of the language [2, 7, 42, 47]. These tools and techniques have in common that the semantics of the object language are formulated in an existing (formal) language with well-understood, executable semantics. The first use of definitional interpreters is by Reynolds, employing them as a vehicle for analysing languages [37, 38]. His analysis took advantage of the formal similarity between denotational and interpretative semantics [39]. Various approaches to formal semantics can be explained in terms of Initial Algebra Semantics [12] in which algebraic signatures denote the constructs of a language and semantics are expressed as algebras over signatures. Modular approaches have been developed that make it possible to extend languages with little or no overhead [44], such as monad transformers [25, 21], algebraic effect handlers [33, 49], entity propagation in Modular Structural Operational Semantics [26, 2], and copy-rules and forwarding in Attribute Grammars [46, 43]. These approaches greatly enhance the practice of defining and maintaining definitional interpreters. In modern general-purpose languages, we see advanced use of monads in Haskell [23, 31], Object Algebras [29] in Java, C# and Scala and intrinsically-typed definitional interpreters in Agda [40].

The usage of an execution graph that contains all configurations produced through program execution is related to back-in-time debugging [22, 20, 9, 35], in which programmers can go 'back in execution history'. The execution graph, however, captures all components required to reconstruct the full interactive session as it also records the executed programs.

Jupyter is an open-source project for bringing web-based computational notebooks to a wide audience [18]. The Jupyter platform provides a protocol for connecting notebooks to the language kernels, such as IPython and IJava, that take care of program execution. Jupyter is popular and the community supports a large number and wide variety of languages. Within the Jupyter platform, the exploring interpreter algorithm can serve as a layer on top of language kernels to improve support for exploratory programming within Jupyter notebooks.

4 Implementation

This section presents and discusses a generic implementation of the exploring interpreter algorithm of [8] in Haskell using a WHILE language as an example.

```
data Command = Seq Command Command
             | Assign String Expr
             | Print Expr
             | While Expr Command
             | Skip
data Expr    = Leq Expr Expr | Plus Expr Expr | LitExpr Literal | Id String
data Literal = LitBool Bool | LitInt Integer

whileInterpreter :: Command → Config → Config
data Config  = Config { cfgStore :: Store, cfgOutput :: Output }
type Store   = Map String Literal
type Output  = [String]
initialConfig = Config { cfgStore = empty, cfgOutput = [] }
```

The definitions of *Command*, *Config*, *initialConfig* and *whileInterpreter* form a language according to Definition 1. The definitional interpreter (not shown) uses configurations with lists of strings as output and stores to record assignments.

An exploring interpreter is implemented as a parameterized data type, where the type parameters denote the programs and configurations of a given language:

```
data Explorer p c = Explorer { defInterp    :: p → c → c
                             , config       :: c
                             , execEnv      :: Gr Ref p
                             , currRef      :: Ref
                             , genRef       :: Ref
                             , cmap         :: IntMap c
                             , sharing      :: Bool
                             , backTracking :: Bool }
```

The *defInterp* field holds the interpreter responsible for executing programs. The *config* field stores the current configuration, i.e. the configuration to be used for the execution context of the next program. The *execEnv* field holds the current execution graph and is implemented as an edge-labelled graph using the `fgl` library[5]. Edges are labelled by programs. The nodes of the execution graph are references (of type *Ref*) to configurations rather than actual configurations. References are implemented as integers and every new configuration gets a unique reference from an increasing counter (using *currRef* and *genRef*). The field *cmap* records the configuration to which each existing reference refers. The field *sharing* determines whether to detect that a configuration has been reached that has already been encountered in which case no fresh reference is generated. With sharing, a configuration is referred to by at most one reference and a node in the execution graph may have multiple incoming edges. Without sharing, multiple references may refer to the same configuration and each node of the execution graph has at most one incoming edge, i.e. the execution graph forms a tree. The *backTracking* field indicates whether a revert action is destructive and deletes nodes and edges.

A smart constructor is defined that, given a definitional interpreter and an initial configuration, produces an *Explorer*.

```
mkExplorer :: Bool → Bool → (p → c → c) → c → Explorer p c
mkExplorer share backtrack interpreter conf = Explorer
  { sharing      = share
  , backTracking = backtrack
  , defInterp    = interpreter
  , config       = conf
  , genRef       = 1
  , currRef      = 1
  , cmap         = IntMap.fromList [(1, conf)]
  , execEnv      = mkGraph [(1, 1)] [] }
mkExplorerStack  = mkExplorer False True
mkExplorerTree   = mkExplorer False False
mkExplorerGraph = mkExplorer True False
```

The smart constructor has additional parameters to determine whether the constructed *Explorer* should apply sharing and (destructive) backtracking. Additional smart constructors are defined that construct *Explorer* variants based on different choices for the *share* and *backtrack* parameters. Without sharing and with destructive reverts, the execution graph forms a linked list with stack-like operations. Without sharing and without destructive revert, the execution graph forms a tree. Section 5 discusses these properties further.

An *Explorer* for the WHILE language can then be obtained as follows:

```
type WhileExplorer = Explorer Command Config
whileTree = mkExplorerTree whileInterpreter initialConfig
```

[5] `https://hackage.haskell.org/package/fgl`

The exploring interpreter algorithm of [8] describes three actions that can be performed on exploring interpreters: **execute**, **revert** and **display** for executing programs, reverting to previous configurations and displaying the execution graph.

The **execute** action applies the underlying interpreter on a given program to transition from the current configuration to a (possibly new) configuration.

```
execute :: (Eq c, Eq p) ⇒ p → Explorer p c → Explorer p c
execute p e = updateConf e (p, defInterp e p (config e))

updateConf :: (Eq c, Eq p) ⇒ Explorer p c → (p, c) → Explorer p c
updateConf e (p, newconf) =
  if sharing e
    then case findRef e newconf of
      Just (r, _) →
        if hasLEdge (execEnv e) (currRef e, r, p)
          then e          {config = newconf, currRef = r}
          else e          {config = newconf, currRef = r
                          , execEnv = insEdge (currRef e, r, p) (execEnv e)}
      Nothing → addNewPath e p newconf
    else addNewPath e p newconf
```

The resulting configuration becomes the current configuration and the *Explorer* components are updated. If sharing is disabled, a configuration is always seen as unique and a new reference is created, the configuration is added to the execution graph, an edge from the original configuration to the new configuration is created, and the association between the new reference and configuration is stored. However, if sharing is enabled and the resulting configuration has already been encountered, then the previously assigned reference is used as the target of the new edge.

The **revert** operation takes a reference and changes the current configuration to the configuration matching the reference:

```
revert :: Explorer p c → Ref → Maybe (Explorer p c)
revert e r = case IntMap.lookup r (cmap e) of
               Just c  | backTracking e → Just e {execEnv = execEnv', config = c
                                                 , cmap = cmap', currRef = r}
                       | otherwise      → Just e {currRef = r, config = c}
               Nothing                  → Nothing
  where
    nodesToDel = reachable r (execEnv e) \\ [r]
    edgesToDel = filter toDel (edges (execEnv e))
                   where toDel (s, t) = s ∈ nodesToDel ∨ t ∈ nodesToDel
    execEnv'   = (delEdges edgesToDel ∘ delNodes nodesToDel) (execEnv e)
    cmap'      = deleteMap nodesToDel (cmap e)
```

If a reference is given without a corresponding configuration, Nothing is returned. If there is a corresponding configuration, then the current reference is changed to the given reference and the current configuration is updated accordingly. Further behaviour of **revert** is determined by the *backTracking* field, indicating whether

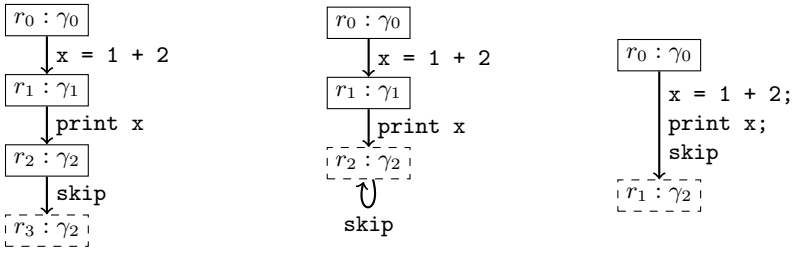

Fig. 4. Execution graphs after executing the WHILE commands `x = 1 + 2`, `print x`, and `skip` without and with sharing, and as a single command respectively. The current node is dashed. The notation $r : \gamma$ denotes a node labelled with reference r referring to configuration γ.

the action is destructive. If it is destructive, then all nodes and edges reachable from the given reference are removed from the execution graph.

Operation **display** produces a structured representation of the execution graph, with the current configuration highlighted. The goal of the display function is to allow interfaces to display and export (parts of) the graph, e.g. to provide an overview, selecting nodes to revert to and saving sessions for later reproduction. To accommodate a wide variety of interfaces, we export several functions for accessing (parts of) the execution graph. For example, to access the entire execution graph, we export the following function:

$$executionGraph :: Explorer\ p\ c \to (Ref, [Ref], [((Ref, c), p, (Ref, c))])$$

The result contains the current node, a list of all nodes and a list of all edges in the execution graph. The edges contain both the reference and the referenced configuration of a node.

To obtain only part of the execution graph we export the following functions:

$$\begin{aligned} &getTrace\ \ :: Explorer\ p\ c \to [((Ref, c), p, (Ref, c))] \\ &getTraces :: Explorer\ p\ c \to [[((Ref, c), p, (Ref, c))]] \end{aligned}$$

These functions provide one or multiple paths – referred to as traces – from the root node to the current node. As discussed in more detail in the next section, a node might have more than one trace (only) when sharing is enabled.

As an example of using exploring interpreters, consider the following sequence of WHILE commands: `x = 1 + 2; print x; skip`. Figure 4 shows the execution graph (with and without sharing) produced when each command in this sequence is executed individually by the exploring interpreter. The first commands adds the assignment of literal 3 to identifier x to the store and gives rise to the node with reference r_1. The second extends the output in the configuration with the literal 3, resulting in the node with reference r_2. The `skip` command has no effect on the configuration. Without sharing a new reference is created nonetheless (reference r_3 on the left of Figure 4). With sharing a self-edge labelled with `skip` is created at the node with reference r_2 (middle of Figure 4).

Folding and unfolding sequences The sequence `x = 1 + 2; print x; skip` can also be executed as a single command, resulting in a single edge from r_0 to r_1 (right of Figure 4). Because WHILE is a sequential language, both interpretations are equivalent in that they yield the same final configuration (γ_2). However, as shown by Figure 4, the resulting execution graphs differ significantly, and, depending on the situation, one execution graph might be preferred over the other. Some interfaces might let the programmer determine which execution is chosen. The following function is introduced to offer the flexibility of choice:

$$executeAll :: (Eq\ c, Eq\ p) \Rightarrow [p] \rightarrow Explorer\ p\ c \rightarrow Explorer\ p\ c$$
$$executeAll = flip\ (foldl\ \$\ flip\ execute)$$

If a program is a sequence of multiple programs to be executed individually, then then the program can be unfolded to produce a list of programs. Conversely, if a list of programs is to be executed as a single program, the list can be folded.

5 Evaluation

In this section, we apply our implementation to two languages – eFLINT and Funcons-beta – and use the resulting specialised exploring interpreters to perform a qualitative evaluation on the generic implementation. The evaluation investigates the impact of destructive backtracking and sharing on the interactions with the execution graph. The result is a discussion on various aspects of exploratory programming, including exploratory programming styles, handling input/output and reproducibility. As part of the evaluation, several extensions to the implementation of the previous section are discussed. The basic and extended implementations are available on Hackage[6]. The specialised exploring interpreters for eFLINT[7] and Funcons-beta[8] are also available online.

The eFLINT language is a domain-specific language (DSL) for formalising norms from a variety of sources such as contracts, regulations and business policies [6]. The language currently has three main uses: exploring a policy specification in order to extend it or improve its internal consistency, statically assessing concrete scenarios for compliance, and dynamically enforcing norms in, and assessing the compliance of, (distributed) software systems. The eFLINT language comes with three interfaces to support these tasks, each built on top of the exploring interpreter for the language: a command-line REPL, a web-interface and a TCP server. The language has been extended to a sequential variant by applying the methodology of [8] and the resulting definitional interpreter is used to specialize the generic exploring interpreter developed in this paper. Figure 5 shows a simple interaction with the command-line REPL in which a fact-type `admin` is introduced to record admin rights of users. The command-line REPL uses non-destructive reverts and sharing. A configuration contains a knowledge

[6] `https://hackage.haskell.org/package/exploring-interpreters`
[7] `https://gitlab.com/eflint/haskell-implementation`
[8] `https://github.com/plancomps/funcons-tools`

```
#1 > Fact admin
New type admin
#2 > +admin(Alice)
+admin("Alice")
#3 > +admin(Bob)
+admin("Bob")
#4 > :revert 2
-admin("Alice")
-admin("Bob")
#2 > +admin(Bob)
+admin("Bob")
#5 > +admin(Alice)
+admin("Alice")
```

```
#4 > :session
#1
|
‘- #2
   |
   +- #3
   |  |
   |  ‘- #4
   |
   ‘- #5
      |
      ‘- #4
#4 >
```

Fig. 5. A session in the command-line REPL for eFLINT.

base of facts and after every **execute** and **revert** action the effects on the knowledge base are shown. The `:session` command shows all the traces in the execution graph in the form of a tree.

Figure 6 shows a part of the eFLINT web-interface in which a single trace is displayed (obtained via *getTrace*). The web-interface uses destructive backtracking and does not use sharing. The current node therefore has exactly one trace. The back-end is provided by a HTTP server built on top of the TCP server. The web-interface is used by first loading a specification file and then submitting a scenario – a sequence of statements and queries – for execution (using the ‘Send phrase’ button). The effects of statements and queries are shown in green and orange in the displayed trace. Violations are shown in red. A state can be expanded (state 8 in the example) to show the contents of the knowledge base and the last statement in the scenario that produced this state. The buttons below state 8 allow the trace to be updated in various ways by translating button-clicks to combinations of **execute** and **revert** actions.

The TCP server is also used to integrate the specialised exploring interpreter as a reasoning engine in multi-agent and service-oriented systems. Components of such systems can interact with one or more instances of the exploring interpreter to learn dynamically about permissions, obligations and violations. As such, eFLINT can be used for dynamic policy enforcement and compliance checking.

The PLanCompS project[9] has identified an open-ended library of so-called fundamental constructs (funcons) that can be used to give a component-based semantics to languages across language paradigms [10, 27]. The funcons have their semantics formally defined in I-MSOS [28] and their I-MSOS specifications are translated to micro-interpreters [7, 5]. These micro-interpreters can be composed arbitrarily to form (definitional) interpreters for different funcon libraries. Funcons-beta is the language defined by the definitional interpreter formed by composing the micro-interpreters of the funcons in the published fun-

[9] `http://plancomps.org`

Fig. 6. A web-interface for eFLINT showing (part of) a trace. State 8 is expanded.

```
#1> bind("input", read)
> "Hello world"
#2> print(bound("input"))
Hello world
#2>
```

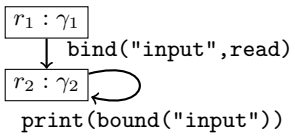

Fig. 7. A session in the command-line REPL for Funcons-beta.

cons library[10]. Figure 7 shows the command-line REPL for Funcons-beta built on top of the specialised exploring interpreter for the language. This exploring interpreter is the result of a small language extension in which Funcons-beta is defined as a sequential language using the *accumulate* funcon as the composition operator $\otimes$. As a result, bindings produced by executing one funcon term propagate to the next. The first funcon term executed in Figure 7 produces a binding for the identifier `"input"`.

Applying the generic exploring interpreter of this paper to these languages required in the order of 50 to 100 lines of Haskell code. In both cases the main effort was defining the definitional interpreter as an extension of the existing interpreter of the language, which involved carefully choosing the contents of the propagated configuration and the method of handling output.

Handling Input/Output, Side-effects and Errors Following the definition of languages (Definition 1), the implementation of the previous section considers a

[10] `https://plancomps.github.io/CBS-beta/Funcons-beta/Funcons-Index/`

definitional interpreter as a pure function expressing the effects of a program on an input configuration. This approach requires the *simulation* of input and output. For example, output can be considered an ever-growing list of (string) values stored in the configurations, as shown by the definitional interpreter for WHILE in Section 4. This choice reduces the potential for sharing since sharing can only take place in between two print statements (discussed further below). Similarly, input can be considered an ever-shrinking list of (string) values with the original input set in the initial configuration.

In Funcons-beta, the *read* reads a value from standard-in as shown in Figure 7. In this example, the program `print(bound("input"))` creates a self-edge because output is not part of the configuration and the program has no other effect. The Funcons-beta command-line REPL takes advantage of an implementation of the exploring interpreter algorithm in which the definitional interpreter can perform effectful computations in a monad, i.e. it has the type:

$$\mathit{defInterp} :: \mathit{Monad}\ m \Rightarrow \mathit{programs} \rightarrow \mathit{configs} \rightarrow m\ \mathit{configs}$$

The command-line REPL for Funcons-beta instantiates m to the IO monad for interacting with standard-in and standard-out.

The introduction of the monad component has additional advantages. In particular, the monad enables distinguishing between effects and side-effects, with side-effects not being recorded in the execution graph. However, side-effects influence the soundness of the wider approach as the implementation can no longer guarantee that executing a program p in the context of configuration γ yields the same result every time. This has a negative impact on the reproducibility of a session and on the soundness of certain graph operations and optimisations.

The execution trace of Figure 6 shows output messages indicating the success of queries and the occurrence of violations. When an eFLINT code fragment is executed (via the 'Send phrase' button at the top), the trace can either be updated using DOM manipulation or the page can be refreshed in its entirety. Although not efficient, refreshing is a convenient way to ensure consistency between the front-end and the back-end, as the front-end is redrawn based on the state of the back-end. This then requires the back-end to record output in order to inform the front-end of the output of programs (such as the results of queries) without re-executing programs. To support the reproducibility of output, we have chosen to add an output component to the definitional interpreters:

$$\mathit{defInterp} :: (\mathit{Monad}\ m, \mathit{Monoid}\ \mathit{out}) \Rightarrow \mathit{programs} \rightarrow \mathit{configs} \rightarrow m\ (\mathit{configs}, \mathit{out})$$

In accordance with Modular Structural Operational Semantics (MSOS) [26, 28], we generalise output to the class of monoidal types, allowing output to concatenate in between executions. Any output produced by the definitional interpreter is stored on the edges of the execution graph, alongside the program producing that output. The updated definitions of *Explorer* and *execute* are as follows:

$$\begin{array}{l}
\textbf{data}\ \mathit{Explorer}\ p\ m\ c\ o\ \textbf{where} \quad \text{-- using GADT extension} \\
\quad \mathit{Explorer} :: (\mathit{Eq}\ p, \mathit{Eq}\ c, \mathit{Monad}\ m, \mathit{Monoid}\ o) \Rightarrow \\
\qquad \{\, \mathit{defInterp} :: p \rightarrow c \rightarrow m\ (\mathit{Maybe}\ c, o), ... \} \rightarrow \mathit{Explorer}\ p\ m\ c\ o
\end{array}$$

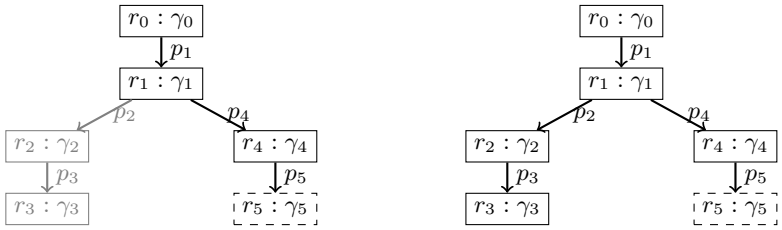

Fig. 8. Execution graph after execution p_1, p_2, p_3, reverting to r_1 and executing p_4, p_5. The gray nodes and edges are removed if the revert action is destructive.

```
execute :: (Eq c, Eq p, Monad m, Monoid o) ⇒
           p → Explorer p m c o → m (Explorer p m c o, o)
execute p e = do (mcfg, o) ← defInterp e p (config e)
                 case mcfg of Just cfg → return (updateConf e (p, cfg, o), o)
                              Nothing → return (e, o)
```

As before, the *updateConf* function is responsible for the extension of the execution graph, now also storing the output on edges. The *Maybe* component of the definitional interpreter is added to support interpreters that may fail. If the definitional interpreter returns *Nothing*, then no changes are made to the execution graph. The interpreter for Funcons-beta fails due to runtime errors, e.g. caused by unbound identifiers. The interpreter for eFLINT performs type-checking to find typing errors and perform coercions. Both types of errors cause the interpreter to fail and yield error messages as part of the output.

Discussions on Backtracking The decision to revert destructively by removing nodes and edges from the execution graph has practical and usability-related consequences. Non-destructive reverts enable a more powerful form of exploratory programming. Consider the two execution graphs in Figure 8, created with and without destructive backtracking. The figure shows how destructive backtracking ensures that there is always exactly one node in the graph without outgoing edges. In other words, exploration always proceeds along a single path and a revert action always undoes the last n changes along that path (for some n). Conversely, when revert is not destructive, multiple paths are explored simultaneously and strategies like depth-first or breadth-first exploration are possible.

Destructive reverts save space by reducing the size of the execution graph. Applications in which multi-path exploration is not required should therefore be able to use destructive reverts. An example of such an application is the execution of a large test-suite in which all tests share a common prefix containing, for example, a number of declarations and initialisation statements. In this case, a programmer can execute all tests by executing the prefix once and subsequently executing all tests of the test-suite with backtracking in between tests to undo the changes of the previous test. Executing a test-suite this way can potentially save large amounts of time while the use of space is reduced with destructive

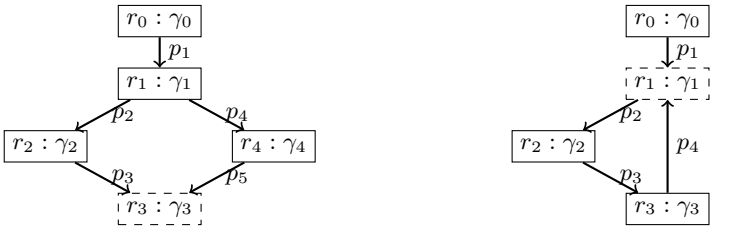

Fig. 9. Execution graphs showing convergence (left) and a cycle (right).

reverts. Owing to the implementation presented in this paper, the eFLINT TCP server interface can be used to execute test-suites in the way described.

We conclude that both destructive and non-destructive reverts should be made available to the interface developer on a per application basis. In fact, we also make a version of *revert* available in which a parameter determines whether the revert is destructive as part of the function call. After all, even when multi-path exploration is desired, a programmer might still wish to undo programs.

Discussions on Sharing The decision to apply sharing – i.e. ensuring that every configuration is referred to by at most one node – has significant impact on the practicality and usability of the exploring interpreter. The execution graph is more space-efficient with sharing rather than without, benefiting especially those applications in which output is not stored in configurations (see Figure 7 and the discussion on output above). However, detecting opportunities for sharing is costly as it requires comparing (possibly many) configurations for equality. Our implementation determines that the type of configurations used by a language must be an instance of the *Eq* type-class. The *Eq*-instances derived by Haskell compilers use structural equality, a costly operation on large datastructures. Moreover, structural equality cannot be used when configurations store functions (such as continuations), in which case a custom equality instance is necessary. This is the case for Funcons-beta, in which a function for reading input (using either real or simulated input) is propagated throughout the definitional interpreter. As this function does not change in between calls to **execute**, it is safe to ignore the function when attempting sharing.

Besides space-efficiency, two further advantages of sharing can be observed. Firstly, through sharing, the exploring interpreter automatically detects the convergence of two exploration paths. In certain applications it will be insightful to the programmer to become aware of convergence. Similarly, sharing will detect cycles. The (abstract) execution graphs of Figure 9 give examples of convergence (left) and a cycle (right). The session in Figure 5 is a concrete instance of the graph showing convergence in Figure 9. Note that by performing effects in a monad, the insights gained from convergence are reduced because convergence only concerns the effects represented by modifications to configurations.

For the second example of a possible advantage of sharing, consider the situation in the graph on the right-hand side of Figure 9 in which r_1 is the current

node. If a program p_5 is to be executed next, and if p_5 is equivalent to p_2, then the exploring interpreter can recognise this and jump to r_2 without executing p_5 (but with adding the edge labelled p_5). If p_5 is a costly program to execute, considerable running time might be saved. This optimisation does not depend on sharing; the same situation could arise if the programmer reverted from r_3 to r_1 (without executing p_4 and without destructive backtracking). However, with sharing, opportunities to apply this optimisation are likely to increase in frequency. To further increase the potential of this optimisation it is beneficial to apply normalisation techniques to programs. The implementation and analysis of this optimisation is left as future work.

A disadvantages of sharing is that the revert action becomes ambiguous because, with sharing, a node can have more than one incoming edge and more than one trace. Selecting a node in the execution graph is not sufficient to revert to a particular moment in time with a unique history of prior actions. A possible solution is to retain a history of actions in the exploring interpreter. Similarly, it is unclear what the effect of a destructive revert should be in the context of sharing. In the current implementation, all outgoing paths of the new current node are removed from the execution graph[11]. Sharing also allows cycles that generate infinitely many paths with a repeated infix. These disadvantages demonstrate that sharing significantly complicates the execution graph in a way that makes it harder for the programmer to align their own mental model with the execution graph.

Although our implementation continues to support sharing, we expect that an exploring interpreter without sharing is sufficient for exploratory programming in many applications. This especially because even without sharing, convergent and cyclic exploration can still be detected by monitoring whether there are configurations referred to by more than one reference. Our implementation does not export a variant of *execute* with a parameter to determine whether to applying sharing. This is to preserve the aforementioned properties of the execution graph, e.g. that the execution graph forms a tree without sharing.

Saving and loading sessions The execution graph of a pure exploring interpreter provides enough information to support the storing and reproduction of sessions generically. One possibility is to export the current configuration, giving the programmer the option to start a new session with the exported configuration as the initial configuration. To also record history, the path from the initial configuration to the current configuration can be exported (i.e. using *getTrace*). When sharing is enabled, all paths from the root node to the current node can be exported (using *getTraces*).

Exporting paths can be done in two ways, affecting in particular the size of the export and the costs of loading a session. The export can contain all components of the path – configurations, references, edges, programs and output – making it possible to load a session without executing programs. However, the soundness of this operation relies on the exploring interpreter being pure; if

[11] With the exception of the node itself, in case of a cycle.

the programs of the saved session were executed in a monad, then there is no guarantee that the context provided by the monad is the same when the session is loaded (e.g. changes in database or file-system). Alternatively, space can be saved by exporting just the sequence of programs labelling the edges on the path. The session can then be loaded by executing this sequence of programs. Assuming the object-language is sequential, this sequence can be folded into a single program as part of the export or as part of loading the session (see the discussion on folding and unfolding in Section 4). Note that in this case, the export is a syntactically valid program that can also be executed with other implementations of the language (e.g. compilers and interpreters).

Finally, the execution graph can be exported in its entirety so that the entire session can be restored.

Discussion In this section we have discussed several extensions to the implementation described in Section 4 and demonstrated the application of the generic back-end to develop several types of interfaces and applications. Based on this, we argue that the generic implementation can be applied widely. For example, we have made no assumptions about the style of (exploratory) programming provided by interfaces. This is demonstrated best by the various types of applications in which the same exploring interpreter for eFLINT is used. The back-end is also applicable to a large class of languages, including at least all languages that can have their semantics expressed as a (pure) transition function. This is best demonstrated by Funcons-beta, which captures the semantics of language constructs across paradigms such as functional programming, imperative programming, procedural programming, object-oriented programming and meta-programming [4]. Our implementation also ensures reproducibility, an important feature in notebooks [18, 32]. In future work we wish to experiment with optimisations in the back-end and generic front-end components.

6 Conclusion

This paper presents a generic back-end for exploratory programming. The back-end is formed by the application of a generic exploring interpreter to a definitional interpreter for the chosen object language. The exploring interpreter adds a level of indirection that makes it possible to deliver multiple programming interfaces for the same language by reusing the back-end and to deliver generic programming interfaces that can be reused across languages. We have performed a qualitative evaluation on the implementation and demonstrated that the back-end can support various styles of exploratory programming, types of interfaces and types of applications such as command-line REPLs, computational notebooks and servers (e.g. to develop web-applications or multi-agent systems). The presented work marks just one step in a bigger research effort aimed at developing tooling and an infrastructure for the independent, modular and reusable design and implementation of programming interfaces for incremental programming and exploratory programming.

Acknowledgements The work in this paper has been partially supported by the Kansen Voor West EFRO project (KVW00309) *AMdEX Fieldlab*, the NWO project (628.009.014) *Secure Scalable Policy-enforced Distributed Data Processing* (SSPDDP) and has been executed in a collaboration with the Agile Language Engineering (ALE) team[12].

References

1. Astesiano, E.: Inductive and operational semantics. In: Neuhold, E., Paul, M. (eds.) IFIP State-of-the-Art Reports, Formal Descriptions of Programming Concepts, pp. 51–136. Springer (1991)
2. Bach Poulsen, C., Mosses, P.D.: Generating specialized interpreters for modular structural operational semantics. In: 23rd International Symposium on Logic-Based Program Synthesis and Transformation, pp. 220–236. Springer (2014)
3. Beth Kery, M., Myers, B.A.: Exploring exploratory programming. In: 2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). pp. 25–29 (2017). https://doi.org/10.1109/VLHCC.2017.8103446
4. van Binsbergen, L.T.: Funcons for HGMP: the fundamental constructs of homogeneous generative meta-programming (short paper). In: Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts & Experience. GPCE 2018 (2018). https://doi.org/10.1145/3278122.3278132
5. van Binsbergen, L.T.: Executable Formal Specification of Programming Languages with Reusable Components. Ph.D. thesis, Royal Holloway, University of London (2019)
6. van Binsbergen, L.T., Liu, L., van Doesburg, R., van Engers, T.: eFLINT: A domain-specific language for executable norm specifications. In: Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences. GPCE 2020, ACM (2020)
7. van Binsbergen, L.T., Mosses, P.D., Sculthorpe, N.: Executable component-based semantics. Journal of Logical and Algebraic Methods in Programming **103**, 184–212 (feb 2019). https://doi.org/10.1016/j.jlamp.2018.12.004
8. van Binsbergen, L.T., Verano Merino, M., Jeanjean, P., van der Storm, T., Combemale, B., Barais, O.: A Principled Approach to REPL Interpreters, pp. 84–100. ACM (2020). https://doi.org/10.1145/3426428.3426917
9. Bousse, E., Leroy, D., Combemale, B., Wimmer, M., Baudry, B.: Omniscient debugging for executable DSLs. Journal of Systems and Software **137**, 261–288 (2018)
10. Churchill, M., Mosses, P.D., Sculthorpe, N., Torrini, P.: Reusable components of semantic specifications. In: Transactions on Aspect-Oriented Software Development XII. pp. 132–179. TAOSD 2015 (2015)
11. Erdweg, S., van der Storm, T., Volter, M., Tratt, L., Bosman, R., Cook, W.R., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A., Konat, G., Molina, P.J., Palatnik, M., Pohjonen, R., Schindler, E., Schindler, K., Solmi, R., Vergu, V., Visser, E., van der Vlist, K., Wachsmuth, G., van der Woning, J.: Evaluating and comparing language workbenches: Existing results and benchmarks for the future. Computer Languages, Systems & Structures **44**, 24–47 (2015)
12. Goguen, J.A., Thatcher, J.W., Wagner, E.G., Wright, J.B.: Initial algebra semantics and continuous algebras. Journal of the ACM **24**(1), 68–95 (Jan 1977)

[12] `http://gemoc.org/ale/`

13. Hayes, B.: Thoughts on Mathematica. Pixel **1**(January/February), 28–34 (1990)
14. Hudak, P., Hughes, J., Peyton Jones, S., Wadler, P.: A History of Haskell: Being Lazy with Class. In: Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages. HOPL III, ACM (2007). https://doi.org/10.1145/1238844.1238856
15. Kahn, G.: Natural semantics. In: Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science. pp. 22–39. Springer-Verlag (1987)
16. Kats, L.C.L., Visser, E.: The Spoofax language workbench: Rules for declarative specification of languages and IDEs. In: International Conference on Object Oriented Programming Systems Languages and Applications. pp. 444–463. OOPSLA 2010, ACM (2010). https://doi.org/10.1145/1869459.1869497
17. Klint, P., Storm, T.v.d., Vinju, J.: Rascal: A domain specific language for source code analysis and manipulation. In: Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation. pp. 168–177. IEEE Computer Society (2009). https://doi.org/10.1109/SCAM.2009.28
18. Kluyver, T., Ragan-Kelley, B., Pérez, F., Granger, B., Bussonnier, M., Frederic, J., Kelley, K., Hamrick, J., Grout, J., Corlay, S., Ivanov, P., Avila, D., Abdalla, S., Willing, C., development team, J.: Jupyter notebooks - a publishing format for reproducible computational workflows. In: Loizides, F., Scmidt, B. (eds.) Positioning and Power in Academic Publishing: Players, Agents and Agendas. pp. 87–90. IOS Press, Netherlands (2016). https://doi.org/10.3233/978-1-61499-649-1-87
19. Lazar, D., Arusoaie, A., Şerbănuţă, T.F., Ellison, C., Mereuta, R., Lucanu, D., Roşu, G.: Executing Formal Semantics with the $\mathbb{K}$ Tool, Lecture Notes in Computer Science, vol. 7436, pp. 267–271. Springer Berlin Heidelberg (2012)
20. Lewis, B.: Debugging backwards in time. Computing Research Repository **cs.SE/0310016** (2003), `http://arxiv.org/abs/cs/0310016`
21. Liang, S., Hudak, P., Jones, M.: Monad transformers and modular interpreters. In: 22nd Symposium on Principles of Programming Languages. pp. 333–343. ACM (1995)
22. Lienhard, A., Gîrba, T., Nierstrasz, O.: Practical object-oriented back-in-time debugging. In: European Conference on Object-Oriented Programming. pp. 592–615. Springer (2008)
23. Marlow, S.: Haskell 2010 Language Report (2010)
24. Milner, R., Tofte, M., MacQueen, D.: The Definition of Standard ML. MIT Press (1997)
25. Moggi, E.: Notions of computation and monads. Information and Computation **93**(1), 55–92 (1991). https://doi.org/10.1016/0890-5401(91)90052-4
26. Mosses, P.D.: Modular structural operational semantics. Journal of Logic and Algebraic Programming **60–61**, 195–228 (2004)
27. Mosses, P.D.: Software meta-language engineering and cbs. Journal of Computer Languages **50**, 39–48 (2019). https://doi.org/10.1016/j.jvlc.2018.11.003
28. Mosses, P.D., New, M.J.: Implicit propagation in structural operational semantics. Electronic Notes in Theoretical Computer Science **229**(4) (2009)
29. Oliveira, B.C.d.S., Cook, W.R.: Extensibility for the masses - practical extensibility with object algebras. In: ECOOP 2012 – Object-Oriented Programming. pp. 2–27. Springer Berlin Heidelberg (2012). https://doi.org/10.1007/978-3-642-31057-7_2
30. Peyton Jones, S. (ed.): Haskell 98, Language and Libraries. The Revised Report. Cambridge University Press (2003)
31. Pickering, M., Wu, N., Kiss, C.: Multi-stage programs in context. In: Eisenberg, R.A. (ed.) Proceedings of the 12th ACM SIGPLAN International Symposium on

Haskell, Haskell@ICFP 2019, Berlin, Germany, August 18-23, 2019. pp. 71–84. ACM (2019). https://doi.org/10.1145/3331545.3342597
32. Pimentel, J.F., Murta, L., Braganholo, V., Freire, J.: A large-scale study about quality and reproducibility of jupyter notebooks. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR). pp. 507–517 (2019)
33. Plotkin, G., Pretnar, M.: Handlers of algebraic effects. In: Castagna, G. (ed.) Programming Languages and Systems: 18th European Symposium on Programming. pp. 80–94. ESOP 2009, Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
34. Plotkin, G.D.: A structural approach to operational semantics. The Journal of Logic and Algebraic Programming **60-61**, 17 – 139 (2004)
35. Pothier, G., Tanter, É., Piquer, J.: Scalable omniscient debugging. ACM SIGPLAN Notices **42**(10), 535–552 (2007). https://doi.org/10.1145/1297105.1297067
36. Rein, P., Ramson, S., Lincke, J., Hirschfeld, R., Pape, T.: Exploratory and live, programming and coding. The Art, Science, and Engineering of Programming **3**(1) (2018). https://doi.org/10.22152/programming-journal.org/2019/3/1
37. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. In: Proceedings of the ACM Annual Conference - Volume 2. pp. 717–740 (1972)
38. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. Higher-Order and Symbolic Computation **11**(4), 363–397 (1998)
39. Reynolds, J.C.: Definitional interpreters revisited. Higher-Order and Symbolic Computation **11**(4), 355–361 (1998)
40. Rouvoet, A., Bach Poulsen, C., Krebbers, R., Visser, E.: Intrinsically-typed definitional interpreters for linear, session-typed languages. In: Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP'20). pp. 284–298 (2020). https://doi.org/10.1145/3372885.3373818
41. Rule, A., Tabard, A., Hollan, J.D.: Exploration and explanation in computational notebooks. In: Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems. p. 1–12. CHI '18, ACM (2018)
42. Sewell, P., Nardelli, F.Z., Owens, S., Peskine, G., Ridge, T., Sarkar, S., Strnisa, R.: Ott: Effective tool support for the working semanticist. Journal of Functional Programming. **20**(1), 71–122 (2010). https://doi.org/10.1017/S0956796809990293
43. Swierstra, S.D., Alcocer, P.R.A., Saraiva, J.a.: Designing and implementing combinator languages. In: Advanced Functional Programming, pp. 150–206. Springer Berlin Heidelberg (1999). https://doi.org/10.1007/10704973_4
44. Swierstra, W.: Data types à la carte. Journal of Functional Programming. **18**(4), 423–436 (Jul 2008). https://doi.org/10.1017/S0956796808006758
45. Trenouth, J.: A survey of exploratory software development. The Computer Journal **34**(2), 153–163 (01 1991). https://doi.org/10.1093/comjnl/34.2.153
46. Van Wyk, E., de Moor, O., Backhouse, K., Kwiatkowski, P.: Forwarding in attribute grammars for modular language design. In: Horspool, R. (ed.) Compiler Construction, Lecture Notes in Computer Science, vol. 2304, pp. 128–142. Springer Berlin Heidelberg (2002). https://doi.org/10.1007/3-540-45937-5_11
47. Vergu, V.A., Neron, P., Visser, E.: DynSem: A DSL for dynamic semantics specification. In: 26th International Conference on Rewriting Techniques and Applications, RTA 2015. Leibniz International Proceedings in Informatics, vol. 36, pp. 365–378. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2015)
48. Walicki, M., Meldal, S.: Algebraic approaches to nondeterminism – an overview. ACM Computing Surveys **29**(1), 30 – 81 (Mar 1997)
49. Wu, N., Schrijvers, T., Hinze, R.: Effect handlers in scope. In: Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell. pp. 1–12. Haskell 2014, ACM (2014)