Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Exploratory Programming for the Masses with Support for Polyglot Exploration

Author: Damian Frölich (2584313)

1st supervisor:	Dr L. Thomas van Binsbergen (UvA)
2nd reader:	Dr Clemens Glerck (UvA)

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

August 12, 2021

Abstract

Exploratory programming is a programming technique that intertwines design and implementation, and is used across a wide variety of domains. However, current tooling only supports a limited form of exploratory programming or is focused on the text level, preventing interactivity with a running system. Furthermore, many software projects are written in a combination of languages, but most tooling is only focused on exploration with one language, complicating exploratory programming in such projects.

In this thesis, I present an implementation for a generic exploring interpreter that supports a wide variety of interactive exploratory behaviours, a consistent exploratory experience, and makes exploratory programming available to a broad range of languages. With the implementation, language parametric interfaces for exploratory programming are possible and implemented. These interfaces simplify the creation of interfaces supporting exploratory programming, reducing the effort required by a language to support exploratory environments. Furthermore, a protocol for the exploring interpreter is designed, allowing interfaces to be re-used by many exploring interpreter implementations. Via these interfaces and the exploring interpreter, a method for polyglot exploratory REPLs is presented based on language compositions, enabling polyglot exploratory programming and setting a step into the direction of exploratory language development.

Acknowledgements

First, I want to thank my second reader, Clemens Grelck, for being my second reader during a precious time in his life. Second, I want to thank Ana Oprescu for making this thesis possible by setting up the connection between me and my supervisor. Third, I want to thank my parents for the extensive support, which allowed me to to fully focus on my thesis. Last, but definitely not least, I am forever grateful to my supervisor, Thomas van Binsbergen, for the extensive feedback, the patience when I introduced a new implementation, the fruitful discussions that helped form this thesis, and giving me the opportunity to publish my first paper.

Contents

List of Figures

List of Tables

1

Introduction

Exploratory programming [1, 2] is a style of programming that intertwines system design and implementation. A main characteristic of such an approach is that the goal worked towards is not fully known beforehand, but is formed while exploring with code, constituting an iterative and volatile style of development.

Usage of exploratory programming is seen in a wide range of different domains, where exploratory programming has different goals. For example, Alice [3] provides an environment for children to create 3D animated stories with the goal to teach children programming via an exploration process. The Processing environment [4] is another example, which enables the creation of digital art via exploratory programming. Notebooks, like Jupyter [5], support basic forms of exploratory programming combined with literate programming, and are accessible for non-programmers. Usage of notebooks is seen in domains like data science [2] and computational science [6].

Currently, much exploration is done via text editors [7] or interactive systems [8] like a read-eval-print loop (REPL) or computational notebook.

Text editors offer undo and redo functionality, which is useful when backtracking to earlier version and going back from earlier versions to later version. However, the undo and redo functionality are often linear, a user can only redo the last edit and when editing after an undo, the redo option is lost. In addition, text editors do not provide the interactivity REPLs and notebooks provide, making iterations slower.

While systems like REPLs and notebooks provide interactivity, they often do not support undo and redo functionality [9]. Hence, a user can move forward in an exploration session with such a system, but is unable to have multiple versions of an implementation such that it is easy to switch between different implementations without losing work and compare different implementations, which is a crucial component in the exploratory process [8, 10]

Recent focus on exploratory programming resulted in new tools to support a more extensive form of exploratory programming [11, 12]. More extensive exploratory programming is obtained by allowing different versions or derivations of an implementation and enabling the comparison of these implementations, thus moving away from the linear behaviour seen in most editors.

However, current exploratory tools are mostly focused on the text level, missing interactivity and only enabling shallow inspection of run-time state. In addition, most tools that support interactivity, only support interactivity via one language, but many systems are written in a combination of languages to utilise more libraries, use the right language for a problem, or to achieve performance criteria [13, 14]. Exploratory programming in such environments is difficult, because it requires managing multiple isolated environments. When environments are isolated, a definition introduced in one environment is only available in that environment, requiring a user to perform manual syncing or restarting of environments to share definitions, impeding interactivity and volatility.

In this thesis I investigate techniques to support interactive exploratory programming in an exploratory environment for a wide range of languages, without requiring extensive effort on the language implementation side. Furthermore, I investigate how to facilitate polyglot exploratory programming in a flexible, safe, and low effort form.

Concretely, in this thesis I make the following contributions:

- create a generic implementation of the exploring interpreter model that adds support for exploratory programming to any language following the language definition;
- discuss different styles of exploratory programming enabled by the exploring interpreter implementation;
- introduce an updated model for the exploring interpreter that provides consistent exploratory behaviour and allows impure interpreters and interpreters that can indicate program errors;
- define a language generic protocol for the exploring interpreter and implementations of language parametric interfaces for exploratory programming, making exploratory programming interfaces almost free;
- and introduce a method for language composition, enabling the creation of coarse and fine-grained polyglot REPLs that support exploratory polyglot programming, and opening up experimentation with systems to support exploratory language development.

1.1 Thesis structure

In chapter 2 the necessary background for this thesis is given, including a definition for exploratory programming and the formal model of the exploring interpreter. Background is followed by a brief discussion of related work in chapter 3. chapter 4 introduces and evaluates the generic exploring interpreter. Insights obtained from the evaluation are used to introduce a new exploring interpreter model in chapter 5. Language independent interfaces for exploratory programming are introduced in chapter 6, which is followed by a method for constructing polyglot exploratory programming environments in chapter 7. The presented work is broadly discussed in chapter 8 and a conclusion is presented in chapter 9.

2

Background

2.1 Exploratory programming and interactive environments

Exploratory programming is not a new concept and has been present in interactive environments, like Smalltalk [15], LISP [16] and SELF [17]. Interactive environments make several tools available in an interactive system to the programmer. These tools include, but are not limited to, read-eval-print loop (REPL) tools, profilers, debuggers, and sometimes text editors. The interactivity and tools of these components enable quick iteration and prototyping, and quick insights in the state of an execution, supporting limited forms of interactive exploratory programming.

In this thesis, exploratory programming is placed as a component in an interactive system. As such, interacting with an interactive system is not exploratory programming. Exploratory programming is a separate activity, like debugging, that can be performed in an interactive system. By placing exploratory programming as a separate component in interactive systems, debugging or similar activities are not considered exploratory programming in this thesis. This distinction follows from the definition of exploratory programming given in [2]. The paper defines exploratory programming as a task with two properties.

- (Property 1) The programmer writes code as a medium to prototype or experiment with different ideas.
- (Property 2) The programmer is not just attempting to engineer working code to match a specification. The goal is open-ended, and evolves through the process of programming.

The second property denotes that the goal with exploratory programming is open-ended and evolves as a result of the exploration process. Since debugging has a clear goal, it is not considered exploratory programming. Nonetheless, I do recognise that the tooling provided by the exploratory programming component can be combined with other components in the interactive environment to improve the exploratory experience, decrease iteration time, and increase capabilities of other components. Hence, the exploratory programming component is not an isolated component in the interactive environment.

2.2 Principled approach to REPLs and exploring interpreters

Previous work [9] proposes a principled approach to REPLs. The principled approach describes how to construct a REPL for a language by identifying that the interaction with a REPL follows naturally via sequential languages. In the paper, a language is defined as follows.

Definition 2.2.1. *A language L is a structure $\langle P, \Gamma, \gamma^0, I\rangle$ with P a set of programs, Γ a set of configurations, γ^0 an initial configuration and I a definitional interpreter assigning to each program $p \in P$ a function $I_p : \Gamma \to \Gamma$.*

A language is a sequential language if it follows the following definition.

Definition 2.2.2. *A language $L = \langle P, \Gamma, \gamma^0, I\rangle$ is sequential if there is an operator ; such that for every $p_1, p_2 \in P$ and $\gamma \in \Gamma$ it holds that $p_1; p_2 \in P$ and that $I_{p_1;p_2}(\gamma) = (I_{p_2} \circ I_{p_1})(\gamma)$*

The sequentiality definition describes that there is no difference between submitting isolated programs or submitting sequenced programs, and every isolated program can be sequenced with another isolated program to create a new valid sequenced program. Via this behaviour, evaluation performed by a REPL is not influenced by the submission style of the user, ensuring consistent behaviour.

The paper continues by introducing an exploring interpreter, a bookkeeping device built on top of a definitional interpreter — an interpreter also defining the operational semantics of a language.

Definition 2.2.3. *An exploring interpreter for a language $\langle P, \Gamma, \gamma_0, I\rangle$ is an algorithm maintaining a current configuration (initially γ_0) and an execution graph (initially containing just the node γ_0) and iteratively executing one of the following actions. At any moment the execution graph is a subgraph of the reachability graph from γ_0*

- *execute(p): transition from the current configuration γ to the configuration $\gamma' = I_p(\gamma)$, where $p \in P$ is provided as input, and subsequently:*
  - *add γ' to the set of nodes (if new),*

  - *add $\langle\gamma, p, \gamma'\rangle$ to the set of edges (if new).*

- *revert(γ): take γ as the current configuration for the next action, where $\gamma \in \Gamma$ is provided as input,*
- *display: produce a structured representation of the current graph, distinguishing the current configuration in the graph from the other configurations.*

With the exploring interpreter, the definitional interpreter is extended with support for interactive exploratory programming.

2.3 Data types á la carte

Data types á la carte [18] describes an approach to achieve compositional data types that can be freely combined and new operations over these data types can freely added, solving the expression problem [19].

The expression problem describes the difficulty of extending a data type with new cases and adding new operations over existing data types, while retaining type safety and no recompilation of existing code. In functional languages, the problem arises because it is difficult to extend a data type with new cases. For example, we can encode a simple integer addition language in Haskell as follows.

```
data Expr = Val Int | Add Expr Expr

eval :: Expr → Int
eval (Val v) = v
eval (Add l r) = eval l + eval r
```

Extending the language with a new case, for example adding support for subtraction, requires modification of the *Expr* data type. As a result of modifying this data type, all functions over this data type need to be modified to support the newly added case, requiring recompilation. Alternatively, adding support for a new operation, like pretty printing, is trivial in a functional language: just write a new function over the data type.

To make adding new cases without modification possible, data types á la carte represents data types as functors, which allows combining data types using the co-product of the underlying functor. Using this technique, the expression language can be represented as follows.

```
data Val a = Val Int
data Add a = Add a a

data (f :+: g) e = Inl (f e) | Inr (g e)
```

The *Val* and *Add* data types are now isolated functors, which can be combined using the composition operator (:+:): *Val* :+: *Add*. With this technique, a simple expression can be constructed as follows.

```
Inr (Add 5 5) :: (Val :+: Add) Int
Inl (Val 5) :: (Val :+: Add) Int
```

Since the Add signature is on the right side of the composition, the *Inr* constructor is used to place values of type *Add* into the right side of the composition. For values of type *Val*, the *Inl* constructor is used to place the values on the left side of the composition.

The composition operator takes a type parameter and forwards it to the individual components of the composition, requiring a type argument for a composition. In the example, the *Int* type is given as the argument, which makes *Int* the expected type for the parameters of the *Add* constructor. The *Val* constructor is not influenced by the type parameter, since the *Val* constructor explicitly states the parameter type. However, specifying the type parameter as type *Int* makes it impossible to pass values of the composition type as arguments to the constructor, thus only non-recursive integer expressions can be constructed.

To tie the recursive knot, the approach uses the fixed point of functors [20].

```
data Term f = In (f (Term f))
```

The data type takes a type parameter of kind $* \to *$, corresponding to the kind of a functor. It then passes the *Term f* data type as the argument to the provided type parameter. In our example, the *Val* :+: *Add* composition can be passed as the argument to *Term*: *Term* (*Val* :+: *Add*). As a result, values of type *Term* (*Val* :+: *Add*) can be given to the constructors *Val* and *Add*, making recursive definitions possible.

```
In (Inr (Add (In (Inr (Add (In (Inl (Val 5))) (In (Inl (Val 10)))))) (In (Inl (Val 1)))))
  :: Term (Val :+: Add)
```

The example shows the encoding of the `(5 + 10) + 1` expression, which is a simple expression but results in a non-trivial encoding. To alleviate the construction of terms, the approach defines a typing relation using type-classes. The typing relation is used to perform automatic injections into the composition.

```
class (Functor sub, Functor sup) ⇒ sub :<: sup where
  inj :: sub a → sup a
instance f :<: f where
  inj = id
instance f :<: g where
```

```
    inj = Inl
  instance (f :<: g) ⇒ f :<: (g :+: h) where
    inj = Inr ∘ inj
```

The :<: operator defines a typing relation such that if f :<: g it means that f is subsumed by g, i.e. values of type f can be constructed as part of type g. With the automatic injections, our earlier example can be defined as follows.

```
  inject = In inj
  inject (Add (inject (Add (inject (Val 5)) (inject (Val 10)))) (inject (Val 1))) :: Term (Val :+: Add)
```

which is more concise, and with smart constructors, like *val x* = *inject* (*Val x*), it can be further improved.

Thus, we have a method for combining data types in a functional language without modifying existing data types. However, with the method the ability to define simple functions over data types is lost, because a function must know the structure of the data type it operates on. With compositional data types, the structure is not known and can be freely extended, but the extension is not handled by existing functions.

Nonetheless, because the data types are functors and the co-product of a functor is a functor, catamorphisms — generalisation of folds — can be used to operate on the composition of data types.

```
  foldTerm :: Functor f ⇒ (f a → a) → Expr f → a
  foldTerm f (In t) = f (fmap (foldTerm f) t)
```

The first argument to the *foldTerm* function is called an algebra, denoted by the $f\ a \to a$ type, and defines how the resulting value is constructed.

To enable extension of new cases at the function level, the approach makes algebra's part of type classes. Type classes are used because they provide ad-hoc polymorphism and are open for extension by defining an instance of the type class. In our example, a simple algebra evaluating the expression to their integer value can be defined as follows.

```
  class Eval f where
    evalInt :: f Int → Int
  instance Eval Add where
    evalInt (Add l r) = l + r
  instance Eval Val where
    evalInt (Val v) = v
    -- Lift the algebra to operate on compositions.
  instance Eval (f :+: g) where
    evalInt (Inl l) = evalInt l
    evalInt (Inl r) = evalInt r
```

The *Eval f* class defines the type class with the algebra specification, and data type cases define an instance of the class To allow evaluation of composed data types, an instance is also defined for the composition.

A full implementation of the data types á la carte approach and extensions on the approach are presented in [21]. The extensions include, among others, automatic generation of smart constructors using template Haskell and automatic lifting of algebras to operate on compositions, which removes a lot of boilerplate code. With the library, our earlier example can be rewritten as follows.

```
data Val a = Val Int
data Add a = Add a a
 $ (derive [makeTraversable, makeFoldable,
   makeEqF, makeShowF, smartConstructors]
   ['' Val, '' Add])
class Eval f where
   evalInt :: Alg f Int   -- Internal type for the Algebra of f Int -> Int
instance Eval Add where
   evalInt (Add l r) = l + r
instance Eval Val where
   evalInt (Val v) = v
 $ (derive [liftSum] ['' Eval])
```

The *derive* expressions denote template Haskell expressions and are implemented by the library. The first derive generates automatic injections for the *Val* and *Add* data types, and instances of the *Show*, *Eq*, *Traversable*, and *Foldable* type classes. The second derive automatically lifts the *Eval* algebra to an algebra defined over a composition of data types.

With the generated smart constructors our earlier example of the `(5 + 10) + 1` expression can be written in the following way.

```
expr = (iAdd (iAdd (iVal 5) (iVal 10)) (iVal 1)) :: Term (Val :+: Add)
```

In the example, the functions prefixed by 'i' are the generated smart constructors, which automatically inject the value into the data type composition.

2.4 Polyglot programming

Polyglot programming[1] is a term describing the technique of using multiple programming languages in the same context [22]. This style of programming prevents language lock-in,

[1] http://nealford.com/memeagora/2006/12/05/Polyglot_Programming.html

enabling usage of the right language for a problem, increasing productivity and maintenance; and widens the number of libraries usable in a context, promoting re-usability.

Usage of polyglot programming is seen in many software projects and languages. For example, mixing PHP and HTML is a form of polyglot programming; using a web framework with an SQL database, where SQL statements are encoded as strings; or an embedded regex language in a programming language.

With polyglot programming, a distinction between two types can be made: coarse-grained polyglot programming and fine-gained polyglot programming. In coarse-grained polyglot programming, there is a clear distinction between the languages used in the polyglot system and languages can not be freely mixed, but there is interoperability between the languages. With fine-grained polyglot programming, languages can be mixed arbitrarily, allowing usage of language constructs not present when working with the languages in isolation.

3

Related Work

3.1 Exploratory programming

In the current literature, several systems are introduced that improve the exploratory programming experience compared to ordinary REPLs and notebooks.

Variolite [11] is a code editing tool on top of the Atom editor, with support for exploratory programming. Exploratory programming is supported by enabling a user to define different variants of a piece of code. A variant is created by selecting lines in the editor and wrapping it in a variant. After wrapping the piece of code in a variant, a new variant can be created that replaces the old variant. When replaced, the initial variant is still available in the editor and the user can swap back to the initial variant with the click of a button. Because the tool is as the text level, it is language generic instead of language parametric. Thus, there is no need for a language to implement anything to obtain this form of exploratory programming. Furthermore, variants can be defined inside pieces of code, for example, a variant can be defined on one line in a function, enabling a very fine grained form of exploratory programming. However, currently, the tool provides no integration with the running system of the code. Instead, it runs the file, captures output produced by the program, and stores the output, parameters and input to the program into a file which can be inspected by a user.

A similar tool at the text level is introduced in [12]. Instead of requiring the user to explicitly define a variant, the tool tracks all operations made by a user, assigns the operations a version, and places an indicator in the text editor at the location of the change. Indicators can be used by a user to operate on the text, such as redoing or undoing that specific operation. Operations are local, thus undo only acts on the specific location and not the whole file, as is the case with regular undo in a text editor. Furthermore, the tool uses executions of the code as a delimiter. Edits that are both performed between to executes are marked as dependent and placed in a group, such that operating on one part of the group, operates on the full group.

3.2 Polyglot systems

GraalVM [23] is a virtual machine (VM) with support for high performance polyglot programming [24]. The approach defines a common format via the Truffle framework [25]. Truffle is a framework for implementing abstract syntax tree (AST) interpreters in Java. All languages on the GraalVM implement an AST interpreter in Truffle, making Truffle a bridge between the languages. However, AST interpreters are still language specific and can store their interpreter data in any way required. Consequently, languages can not just use data from another language. Instead, communication between languages is performed via messages. A message denotes an action on an object, like reading a property, and is transformed into a AST interpreter operation by the receiving side. This requires languages to implement the message protocol before a language can be used in a polyglot system. Using the Truffle framework, GraalVM provides polyglot REPLs. Since Truffle is focused on the abstract syntax, the provided polyglot REPLs are coarse grained. Thus, it must be made explicit that another language is used and freely mixing of language constructs is not possible.

Wrattler [26] supports polyglot programming in notebooks. Wrattler uses multiple run-times to evaluate polyglot programs. To share data between the different run-times, Wrattler uses a data store. A data store stores data frames, and run-times obtain input via the data store and place output in the data store. Hence, Wrattler extracts data management from the evaluation performed in a run-time, allowing different run-times to share data. However, a run-time must add support for data frames and communication via the data store, before it can be used in the polyglot notebooks. In addition, because Wrattler uses multiple run-times, one for every language, only coarse grained polyglot programming is possible.

4

Generic Exploring Interpreter

Some of the contributions presented in this chapter are published in the 2021 edition of Trends in Functional Programming [27], and the corresponding paper was awarded the best student paper prize[1]*.*

In this chapter I introduce the initial implementation of a generic exploring interpreter, based on the exploring interpreter model defined in [9]. The introduction is accompanied by a running example in the form of a simple *While* language, which is introduced in section 4.1. After introducing the running example, the generic exploring interpreter implementation is presented and is followed by an evaluation.

4.1 Running example

The running example used alongside our implementation is a simple *While* language as presented in [28]. In Haskell, the *While* language can be defined as follows.

```
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 = [] }
```

[1]http://tfp2021.org/

The *Command*, *Config*, *initialConfig*, and *whileInterpreter* form a language according to Definition 2.2.1. *While* is also a sequential language, because the definitional interpreter is implemented according to Definition 2.2.2.

4.2 Initial implementation

An exploring interpreter is implemented as a parameterised data type, where the type parameters denote the programs and configurations of the 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 }
type Ref = Int
```

via these types, the type of the language specific definitional interpreter (*defInterp*) is inferred. In addition, the exploring interpreter tracks the current configuration (*currRef*) and the history of previous configurations and the transitions between the configurations (*execEnv*). The *execEnv* is an edge-labelled graph as provided by the **fgl** library[1]. In the graph, the labels along the edges denote the executed program between two configurations, and the nodes store references to configurations. A reference, represented by the *Ref* type, is an index in the *cmap* and maps to a concrete configuration. References are unique and generated for every new configuration via *genRef*. Storing references in the execution graph instead of concrete configuration has two advantages: displaying large execution graphs becomes more concise by displaying references instead of actual configurations, and referencing configurations from a text-based interface — like a command-line — becomes feasible. If references are not used, referencing a configuration is impossible and the full configuration must be specified when operating on a configuration. With references, the information inside a configuration is not lost, because an interface can request the configuration corresponding to a reference via the *cmap*. The remaining fields of the explorer type, *backTracking* and *sharing*, denote how the explorer operates when exploring. The *sharing* field influences the execution behaviour of the explorer, and the *backTracking* fields influences the reverting behaviour. The combination of these two fields results in four different

[1] https://hackage.haskell.org/package/fgl

behaviours, as displayed in Table 4.1. In this chapter, the main focus is on the Stack, Tree, and Graph behaviours since those are the behaviours introduced by the original paper.

Table 4.1: The different behaviours of the exploring interpreter for the combinations of sharing and backtracking.

	Backtracking enabled	Backtracking disabled
Sharing enabled	Graph Structured Stack	Graph
Sharing disabled	Stack	Tree

Construction of an explorer requires construction and initialisation of the *cmap*, initialising the *currRef* and *genRef*, and adding the initial configuration to the environment. To ensure the correctness of this process, several smart constructors are provided that, on construction, produce 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
```

Via these smart constructors, an explorer for the *While* language can be constructed.

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

4.2.1 Operations

The exploring interpreter model from [9] describes three actions that can be performed on the exploring interpreter: **execute**, **revert**, and **display**, for executing programs, reverting to previous configurations, and displaying the exploration state, respectively.

The **execute** action is implemented as an execution of the definitional interpreter on the current configuration, with the program to execute as the supplied argument. The result of this execution is a transition from the current configuration to a possibly new configuration for the supplied program.

```
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 processing of a new configuration depends on the *sharing* flag. If *sharing* is enabled, the execution graph is first searched for a configuration equal to the new configuration. If such a configuration is found, an edge between the current configuration and the found configuration is created. If a configuration is not found, a new configuration is created and the execution environment is extended, including an edge from the current configuration to the newly created one. In both cases, the current configuration is updated to point to the configuration transitioned to. When *sharing* is disabled, the behaviour is identical to the behaviour with *sharing* when no equal configuration is found: an extension of the execution environment with a new configuration.

The **revert** action is implemented as an interaction with the explorer influenced by *backTracking*.

```
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)
```

The **revert** action receives a reference as an argument and reverts the current reference to the given reference and updates the current configuration accordingly. If the supplied reference does not correspond with a configuration, *Nothing* is returned — indicating no change in the explorer. When *backTracking* is enabled, the **revert** action is destructive by removing all nodes and corresponding edges reachable from the configuration reverted to.

The **display** action provides a structured representation of the current exploration graph. This is implemented by returning the nodes and edges of the execution graph, and the current configuration.

```
executionGraph :: Explorer p c → ((Ref, c), [(Ref, c)], [((Ref, c), p, (Ref, c))])
```

In this representation, both the reference and the actual configuration are returned, ensuring that all information is present for interfaces to display context dependent information.

Displaying a full execution graph might be infeasible when the exploration reached a sufficient size. In such cases, only displaying the history of the current configuration — also known as a trace — provides an alternative display style.

```
getTrace  :: Explorer p c → [((Ref, c), p, (Ref, c))]
getTraces :: Explorer p c → [[((Ref, c), p, (Ref, c))]]
```

The *getTraces* version is required when *sharing* is enabled, because a configuration might be reached via multiple paths, resulting in multiple traces.

The trace functions operate from the initial configuration. Such a starting point can be inefficient or insufficient for certain display styles. For example, an interface displaying the current and the two previous configurations will request unnecessary information when the trace is large and the trace functions are used. Requesting unnecessary information can be problematic when the interface requests the data over a network, where the time till display is influenced by the amount of data being send. To accommodate different requirements and display styles, the display operations for the path between any two configurations are provided as well.

```
getPathFromTo  :: Explorer p c → Ref → Ref → [((Ref, c), p, (Ref, c))]
getPathsFromTo :: Explorer p c → Ref → Ref → [[((Ref, c), p, (Ref, c))]]
```

The presented functions provide the necessary building blocks to construct more complex and language specific display styles.

Using the provided implementation, a REPL for *While* with exploratory programming can be constructed as follows.

```
repl :: WhileExplorer → IO ()
repl exp = do
  putStr ("\n#" ++ show (E.currRef exp) ++ " > ") ≫ hFlush stdout
  input ← getLine
```

```
case break isSpace input of
  (":display", _) → display exp ≫ repl exp
  (":revert", mint) | Just ref_id′ ← readMaybe (dropWhile isSpace mint)
                    → E.revert ref_id′ exp ≫= repl
                    | otherwise → putStrLn "Integer argument required" ≫ repl exp
  _                 → case whileParse input of
                        Left err → putStrLn err ≫ repl exp
                        Right program → E.execute program exp ≫= showOutput ≫= repl
```

As an example of using an exploring interpreter, consider the interaction with the exploratory REPL of *While* displayed in Figure 4.1. The session starts by assigning the result

```
x = 1 + 2
print(x)
:revert 0
x = 2 + 1
print(x)
:display
```

Figure 4.1: Example of an exploratory session in the *While* REPL.

of the `1 + 2` expression to the variable `x`. The `x` variable is then printed by the `print(x)` command, which is followed by a **revert** to the configuration referenced by 0. After this **revert**, the command `x = 2 + 1` is executed and followed by the `print(x)` command, printing the `x` value. Finally, the exploration environment is displayed via the `display` operation. The results of the `display` operation for the Stack, Tree, and Graph behaviours are displayed Figure 4.2.

Every behaviour results in a different execution environment, indicating that the behaviours influence exploratory programming in different ways, which is further explored and evaluated in the next section.

Since *While* is a sequential language, programs can be sequenced to construct new programs. In the example session, it is possible to execute the `x = 2 + 1;print(x)` as a sequence — thus as one program. In case of the Stack behaviour, this results in the execution graph displayed in Figure 4.3. The exploring interpreter executes the sequenced program as one program — thus one execute operation — resulting in one transition. However, it is also possible to execute the sequenced program with two execute operations, resulting in the original execution environment. The preferred execution graph is difficult to determine generically and depending on the situation, one might be preferred over the other. Therefore, the following function is provided to allow the execution of a sequence of

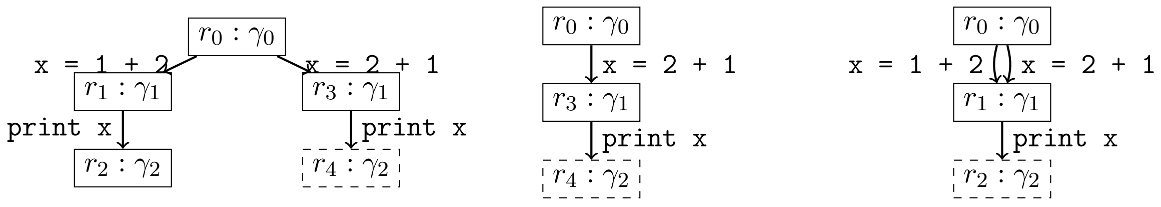


Figure 4.2: Execution graphs for the Tree (left), Stack (middle), and Graph (right) behaviours resulting from the execution of the *While* exploratory session from Figure 4.1. Nodes store the configuration (γ) and the corresponding reference (r), and are uniquely identified by their reference. Edges represent a transition from the source configuration to the target configuration as a result of executing the program — which is displayed as the edge label. The current configurations are represented with dashed lines.

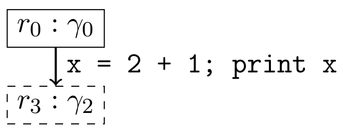


Figure 4.3: Execution graph for the Stack behaviour resulting from the execution of the *While* exploratory session from Figure 4.1 as sequenced program, resulting in only one transition. Nodes store the configurations (γ) and the corresponding reference (r), and are uniquely identified by their reference. Edges represent a transition from the source configuration to the target configuration as a result of executing the program — which is displayed as the edge label. The current configuration is represented with dashed lines.

programs in separate execute operations, giving interfaces the choice to execute a sequence as one program or as separate programs.

$$\begin{array}{l} \mathit{executeAll} :: (\mathit{Eq}\ c, \mathit{Eq}\ p) \Rightarrow [p] \to \mathit{Explorer}\ p\ c \to \mathit{Explorer}\ p\ c \\ \mathit{executeAll} = \mathit{flip}\ (\mathit{foldl}\ \$\ \mathit{flip}\ \mathit{execute}) \end{array}$$

4.3 Evaluation

To evaluate the exploring interpreter implementation, the running example is used together with two new languages — eFLINT and funcons-beta — to construct exploratory environments. The resulting specialised exploring interpreters are used to evaluate the generic implementation.

The eFLINT language is a domain specific language (DSL) for norm formalisation of a variety of sources, such as contracts and business-policies [29]. The language supports three interfaces: a command-line REPL, a web-interface, and a TCP server, which are all operating via the exploring interpreter.

An example session in the eFLINT REPL is given in Figure 4.4. The session defines a new fact called `admin`, which can be used to assign admin rights to users. The fact database can then be queried to determine if a specific user has the correct rights to perform specific actions. The eFLINT REPL shows the effect of a **revert** by displaying the information that is removed due to the revert. In the session, this is visible when the revert is performed to the second configuration, resulting in the removal of the admin rights for Bob and Alice — denoted by the `-admin` output.

Figure 4.5 shows part of the eFLINT web-interface, displaying a single trace that was obtained via the *getTrace* functionality of the exploring interpreter implementation. The eFLINT web-interface supports exploratory programming with the Stack behaviour, and communicates using JSON with a back-end HTTP server built on top of the eFLINT TCP server. With the web-interface, specification files can be loaded that start an exploratory session from a pre-defined point. After, a sequence of statements and queries can be submitted for execution (using the ‘Send phrase’ button). The effects of these statements and queries on the configurations are shown in green and orange in the displayed trace. When a violation occurs, it is marked red. Furthermore, the web-interface allows expansion of states, making it possible to inspect the state of the system at specific points in the exploration process. When expanding, the interface provides several buttons to modify the exploration session. When a user clicks a button, the operation is translated in to combinations of **execute** and **revert**.

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

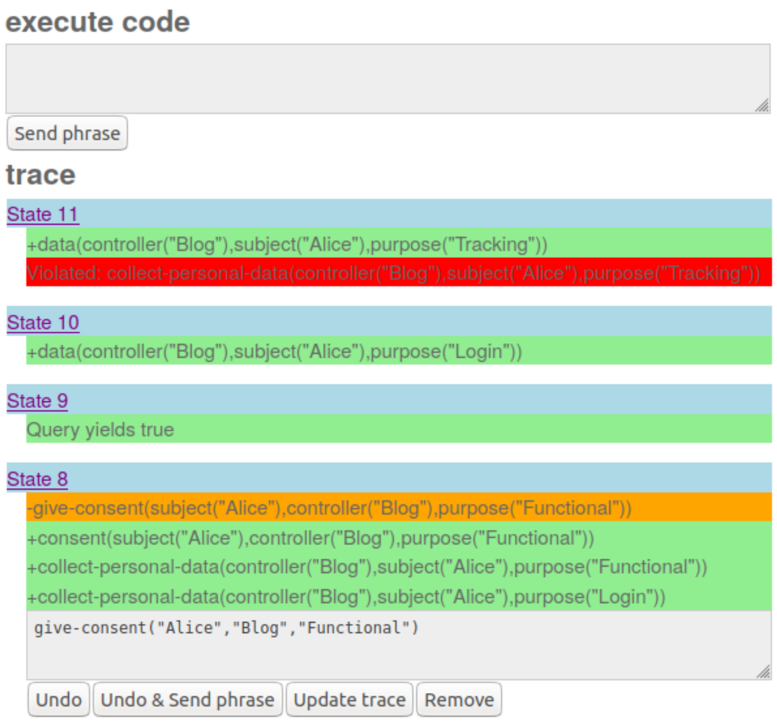
Figure 4.4: An exploratory session — using the Tree behaviour — in the eFLINT command-line interface. On the left the REPL input and output is visible and on the right the exploration environment is displayed.

The TCP server is also used a central component in an integration with the Akka framework to support actor oriented programming with normative actors — actors that manage an eFLINT specification. Actors can communicate with the normative actors by sending eFLINT messages, making different interactions possible. For example, an actor can check via the normative actor if an operation is allowed before performing it.

The PLanCompS project[1] aims to define languages formally using a component-based semantics approach that describes languages in terms of fixed semantics language constructs — shortened as funcons [30, 31]. Funcons have their semantics defined using I-MSOS and are translated into micro interpreters. Micro interpreters can be composed to construct more complex definitional interpreters. An example of a definitional interpreter constructed by composing micro interpreters is funcons-beta, which uses the micro interpreters defined in the funcons library[2]. The funcons library offers a broad variety of different funcons that suffice to define definitional interpreters for languages across paradigms[3], enabling an evaluation of the exploring interpreter over multiple programming paradigms.

Figure 4.6 shows the **display** result of an exploratory session in the funcons-beta REPL operating on an exploring interpreter. The session uses the `read` funcon to read a value

[1]`http://plancomps.org`
[2]`https://plancomps.github.io/CBS-beta/Funcons-beta/Funcons-Index/`
[3]`https://plancomps.github.io/CBS-beta/docs/Languages-beta/index.html`



execute code

trace

State 11

+data(controller("Blog"),subject("Alice"),purpose("Tracking"))

Violated: collect-personal-data(controller("Blog"),subject("Alice"),purpose("Tracking"))

State 10

+data(controller("Blog"),subject("Alice"),purpose("Login"))

State 9

Query yields true

State 8

-give-consent(subject("Alice"),controller("Blog"),purpose("Functional"))

+consent(subject("Alice"),controller("Blog"),purpose("Functional"))

+collect-personal-data(controller("Blog"),subject("Alice"),purpose("Functional"))

+collect-personal-data(controller("Blog"),subject("Alice"),purpose("Login"))

```
give-consent("Alice","Blog","Functional")
```

Figure 4.5: A part of the eFLINT web-interface in an exploratory session on the exploring interpreter.

$$\begin{tikzcd}
\boxed{r_1 : \gamma_1} \arrow[d, "\texttt{bind(\"input\",read)}"] \\
\boxed{r_2 : \gamma_2} \arrow[loop right, "\texttt{print(bound(\"input\"))}"]
\end{tikzcd}$$

Figure 4.6: A session in the command-line REPL for Funcons-beta.

from *standard-in*. This value is bound to the input identifier using the `bind` funcon. This binding is used in the next statement, where the value bound to the input identifier is requested via the `bound` funcon and then printed to *standard-out* with the `print` funcon.

To apply the generic exploring interpreter to the languages, around 50 to 100 lines of Haskell code were required. Most of that code was for the definition of 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.

4.3.1 Backtracking

The **revert** action makes backtracking possible and gives the programmer the means to play with the execution history, providing a more powerful form of exploratory programming compared to existing interactive tools. The implementation provides two forms of reverting: a destructive and a non-destructive form. The destructive form ensures that the execution graph has only one node without an outgoing edge at any time. With the non-destructive form, multiple nodes without outgoing edges can exist. This difference is clearly visible in the example in Figure 4.1, where the left execution graph displays the non-destructive form and the middle execution graph the destructive form. The destructive form has only the right side of the non-destructive execution environment.

The difference between the execution environments occurs because the exploratory session performs a **revert** action in the middle of the session. As a result, the destructive form destroys the nodes reachable from the node reverted to (r_0), which corresponds to nodes r_1 and r_2. As a result, the execution environment for the destructive form contains less information but also requires less space.

Both forms have their use-cases, the destructive form is useful for exploration sessions that are either linear or are guaranteed to not **revert** to destroyed configurations. The non-destructive form is useful for exploratory session that require a lot of switching between obtained configurations, where a switch occurs more than once.

An example use-case that fits the destructive form is the eFLINT TCP server with Akka. Using the destructive form, the actors can explore a system up to a point where a violation happens, investigate the violation and revert to a previous state and continue from there. This makes it possible to explore different system states, including violations, with the actor framework and inspect in detail the evolution of a violation while not requiring an immense amount of memory.

A similar style is batch testing, where there is common prelude that is executed before every test. With the destructive form, the prelude can be the revert target to revert to after executing a test, allowing another test to be executed without having to re-execute the common prelude and without keeping the successful test evolution in memory. The advantage of this approach occurs when a test fails. On failure, the exploratory system can stop and allow the user to inspect the current test evolution and explore the different states achieved during testing.

Furthermore, the destructive form is also beneficial in scratch pad programming [32], where something is quickly tried, as on a scratch-pad, and after trying is immediately

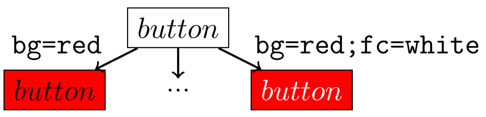


Figure 4.7: A concept for a specialised interface of an exploratory design system.

destroyed. With the destructive form, a common prelude or base for scratch pad programming can be setup, such that a correct environment for scratch pad work is readily available. After the scratch pad session, a destructive revert back to the prelude is performed, making the environment ready for a new scratch pad session without duplicating work or keeping unnecessary state in memory.

Destroying configurations is counter productive when the state of the configuration might be required for further exploration. An example use case of this is given in [7], which performs a case-study on exploratory programming where users use the undo and redo functionality in a text editor. In the case-study, one user uses this combination to compare 16 derivations of one button. With the exploring interpreter, the user could setup a base button style as the main configuration and then execute programs to define the 16 derivations, where every new derivation is preceded by a revert to the main configuration. With the non-destructive form, this would allow the user to have all 16 derivations available at all times, and a specialised interface can visualise the different buttons simultaneously. A concept interface providing this functionality is presented in Figure 4.7.

4.3.2 Sharing

Sharing — ensuring that every configuration is referred by one node — has a similar memory reduction property as the stack behaviour, but achieves it differently. With sharing, the execution environment is based around structural equality of configurations. Programs that affect a configuration identically, transition to the same configuration, resulting in a graph as the execution environment, as showcased in Figure 4.2. As a result, compared to the tree behaviour the number of configurations are reduced without removing information.

Reduction of the number of configurations without losing information is seen in Figure 4.2, where the Graph behaviour contains the same execution history as the Tree behaviour, except the Graph behaviour stores it differently. With the Graph behaviour, there is no difference between the nodes obtained by the execution of the `x = 1 + 2` and `x = 2 + 1` programs from node r_0, since they affect the configuration identically — assign the variable `x` the value 3. Furthermore, the second execution of the `print(x)` program can be skipped, resulting in one less edge in the graph. Skipping an execution means that the definitional interpreter is not called by the exploring interpreter, because the exploring interpreter knows the result of calling the definitional interpreter. In the case of the example, there is already a transition from r_1 to r_2 for the program `print(x)`, when the program is again executed from r_1, the exploring interpreter knows that the result of executing the program is a transition to r_2. Hence, the exploring interpreter can skip the call to the definitional interpreter and present the corresponding configuration, in this case γ_2, as the result. Skipping executions is useful when time expensive programs are repeatedly executed and can be combined with program normalisation to increase the frequency of execution skipping. Skipping execution is, besides the Graph behaviour, also possible in the Tree behaviour. However, it is more likely to occur in the Graph behaviour since it can only occur in the Tree behaviour after a **revert** followed by the execution of a program already executed from the configuration reverted to.

Reduction of execution environment size and higher probability of execution skipping are not the only benefits of sharing. Sharing also introduces extra information not obtained with the Tree and Stack behaviours. Namely, the convergence of programs with respect to the effect they have on configurations. This is seen in the execution environments from Figure 4.2, where the Graph behaviour has two edges leaving node r_0 that both go to node r_1. In this case, the programs belonging to these transitions are `x = 1 + 2` and `x = 2 + 1`. Via the execution environment obtained by the Graph behaviour, it became clear that these two programs affect the configurations identically. This makes it possible to use information obtained from the Graph behaviour to find locally optimal programs.

However, **revert** becomes ambiguous in the context of sharing because a node can have multiple incoming edges, making it impossible to decide the correct history of a **revert** operation. This is problematic in the eFLINT case-study, where the REPL shows the effect of a **revert** by walking the history, as displayed in Figure 4.4. Furthermore, sharing creates infinite paths in case of a cycle. With infinite paths, most **display** operations return infinite results, complicating the interaction with interfaces. Furthermore, with a cycle not all paths are paths created by a user, which can confuse a user when these paths are displayed to the user as their exploration. In addition, the semantics of a destructive **revert** in the context of sharing is unclear. In the implementation, this implemented by removing all reachable edges from the configuration reverted to[1].

Overall, sharing complicates the execution environment and effects the explorer operations, making the interaction with the explorer more complex, but provides higher probability of execution skipping and introduces insights not obtained with the other behaviours.

[1]The node reverted to remains intact in case of a cycle.

4.3.3 Total purity and its implications

The current implementation requires a pure interpreter that always returns a valid configuration, as defined in 2.2.1. These requirements on the definitional interpreter restrict the applicability of the exploring interpreter implementation, because it can not support languages that are not pure nor languages that can have failing interpreters.

Non-pure definitional interpreters occur as soon as a language supports some kind of interaction with the environment, for example, printing to a terminal or communicating with the file system. The funcons-beta language is such a language due to the earlier presented `read` and `print` funcons. Both effects can be simulated. The `read` funcon can be simulated by defining an input sequence before execution, and the `print` funcon can be simulated by storing the output in the configuration and possibly performing the actual output at the end of the execution. However, simulation complicates the exploratory session. Input simulation requires a user to define the input sequence before the execution, which requires a context switch and is not trivial for more complex programs. Output simulation becomes more complex when the simulation performs more complex actions like file writing or network communication where timing or response can play a crucial role for the continuation or correctness of a program. In addition, storing output in configurations affects opportunities for sharing. Output itself is data that in most cases does not affect the execution of subsequent programs. Thus, if two configurations are equal up to their output, a program executes identically when executed from those configurations. However, when output is part of the configuration, sharing can not occur because the output prevents structural equality.

Requiring a definitional interpreter without possibility of error complicates the existence of context specific programs. For example, a program that references an identifier is valid when that identifier is bound to a value and invalid when unbound. With the current definitional interpreter definition, both cases must return a valid configuration. It is unclear what the configuration in case of the invalid program must be, and from the users perspective it is confusing when an invalid program results in a valid configuration and a transition in the exploration environment.

4.3.3.1 Allowing impurity and erroneous computations

To allow languages that perform effects not captured by the model — also known as side-effects — we extend the implementation with an arbitrary monad.

$$defInterp :: Monad\ m \Rightarrow programs \rightarrow configs \rightarrow m\ configs$$

The monad can be instantiated as required, allowing effects to occur outside of the configurations. In case of funcons-beta, m can be instantiated as the IO monad, making real input and output possible.

However, moving effects out of the configurations affects the optimisations the exploring can perform, reduces the extra information obtained from sharing, and complicates the information exchange with certain interfaces.

The earlier mentioned skipping of executions is not possible when a definitional interpreter executes inside a monad, because the effects in the monad are only obtainable by executing the program, making execution skipping also skip effects. Not executing these effects gives an inconsistent view to the user, for example, if a program performs output to a terminal device, this output would not be visible when execution skipping is performed. Such an inconsistent behaviour is unexpected for a user and does not represent the actual execution correctly. Nonetheless, a simplification of the effect monad [33] can be used to create a distinction between pure programs and programs with side-effects. As a result, execution skipping can still be performed on pure programs.

When effects are moved into a monad, the insights obtained from sharing regarding program conversations are reduced, because programs only converge with respect to the effects in the configurations. Thus the convergence seen in an exploration graph states nothing about the side-effects of the programs.

Since monadic effects are not part of the configuration, the information regarding these monadic effects are more difficult to display for certain interfaces. An example of such an interface is the eFLINT web-interface. The web-interface uses the server — including the exploring interpreter — as the source of truth and constructs the interface based on that information. With monadic execution, effects inside the monad are not available in the information provided by the exploring interpreter. As a result, an interface can only display partial information.

Both the convergence of programs and missing information can be solved by extending the definitional interpreter with a monoid component along the result of the definitional interpreter — in accordance with MSOS [34, 35] with respect to output.

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

Via this monoid component, a definitional interpreter can include information that will not be part of the configurations, but instead is stored along the program on the edges — indicating that the values are produced by that program but do not affect the execution of subsequent programs. In case of the eFLINT web-interface, output can become part of the edges making output available to the web-interface without affecting the higher probability of sharing. While shown in the context of output, monoidal values are not restricted to purely output. A language can store any information as long as it forms a monoid. For example, an interpreter can time the execution of a program and store it along the edges, giving some insight to the user regarding execution times.

To accommodate the inclusion of monoidal values, the definitions of *Explorer* and *execute* are updated.

```
data Explorer p m c o where    -- using GADT extension
  Explorer :: (Eq p, Eq c, Monad m, Monoid o) ⇒
    {defInterp :: p → c → m (Maybe c, o), ...} → Explorer p m c o

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)
```

The *updateConf* function is still responsible for updating the execution graph, but now takes monoidal values into consideration. The configuration resulting from the definitional interpreter is wrapped inside a *Maybe* monad, allowing definitional interpreters to fail. While failing, a definitional interpreter can still produce output, allowing for error messages from within the definitional interpreter. This pattern is visible in both eFLINT and funcons-beta. eFLINT performs type-checking for type errors and funcons-beta fails on run-time errors like an unbound identifier. As a result of these errors, both languages yield error messages as part of the output.

By introducing monoidal values and erroneous computations, the semantics of an execution of a sequence of programs is changed, which requires an update to the *executeAll* function.

```
executeAll :: (Eq c, Eq p, Eq o, Monad m, Monoid o) ⇒
  [p] → Explorer p m c o → m (Explorer p m c o, o)
executeAll ps exp = foldlM executeCollect (exp, mempty) ps
  where executeCollect (exp, out) p = do (res, out') ← execute p exp
                                         return (res, out `mappend` out')
```

The new definition still uses a fold, but now executed inside a monad. An **execute** that results in *Nothing* is handled by continuing with the previous well known configuration. Finally, monoidal values are propagated through the sequence, clearly demonstrating the monoidal requirement via the usage of the identity element (*mempty*) and the binary operator (*mappend*).

4.3.4 Sessions

The execution environment contains enough information to support sessions generically. There are different ways to support session that affect the process of saving and loading differently.

Sessions can be supported by exporting one configuration. A user can load this configuration in a new session and continue execution from that point. The method makes session loading quick, since it only requires to read in one configuration. Memory usage in this approach is dependent on the configuration size, but is moderate compared to alternatives. However, the resulting session contains very little information. The full history of the exploratory session is lost, giving no indication how the configuration is obtained.

Alternatively, a session can export a trace by exporting all information — configurations, programs, and output — from the initial configuration to the end of the trace. This provides more context than the previous approach, but also increases the size of the session.

To reduce session size, only programs instead of all information can be exported. A session is then loaded by executing the exported programs, which reconstructs the execution environment. This keeps session size small compared to exporting full configurations, but has the caveat that loading a session requires execution. Execution of programs can increase loading time, especially if the session contains executions that take a considerable amount of time.

With the last two methods, it is also possible to export the full execution environment instead of a trace. Exporting the full environment in most cases results in bigger session and more need for reproducibility.

4.3.4.1 Reproducible sessions

Reproducible sessions fully reconstruct the original environment, including correct references. With reproducible sessions, it is easier to share session, do live collaboration, and make session loading more consistent, simplifying the resumption of an exploratory session.

Exporting all information from an execution environment achieves reproducible sessions in case the definitional interpreter performs no effects in a monad. As stated earlier, effects executed in the monad are unavailable to the exploring interpreter and can therefore not be part of the exported session. However, missing certain information is not the only problem. The environment can change after exporting a session, such that importing the session executes in a different environment. For example, a session that reads in a file has no guarantee that the file exists when reloading the session. Hence, when a definitional interpreter executes inside a monad, it is impossible to guarantee fully reproducible sessions.

Another consequence of monadic execution is sessions altering the environment, for example by writing to a file. When reloading a session that alters the environment, the expected behaviour is not immediately clear. When the session is loaded without program execution, the environment is not changed. Not changing the environment might be described as not reproducing the original session. However, executing programs such that the environment is changed can still not guarantee session reproducibility, because there is no guarantee that the environment is identical to the environment when the session was first constructed. Furthermore, there is a certain security risk when executing code during session loading, especially when sharing sessions — there is no guarantee that a session being shared is fully trusted. The security implications can be partly mitigated by performing passive program execution when loading a session, where the user is questioned per program if it is save to execute. Of course, this makes session loading a more manual process but can help in achieving secure reproducible sessions when side-effects are possible.

Thus, to achieve fully reproducible exploration sessions, the exploring interpreter is dependent on the reproducibility of the underlying environment, and compromises must be made between security and convenience.

5

A Unified Exploring Interpreter

In this chapter I follow up on the insight gained from the evaluation of the generic exploring interpreter implementation. The chapter starts by introducing the unified model — an exploring interpreter model that combines the different behaviours of the initial exploring interpreter — and is followed by an implementation and evaluation of the new model.

5.1 Unified model

Behaviours were a prominent concept in the previous chapter, since they affected functionality of the exploring interpreters resulting in different execution environments for the same session. The unified model — presented in this section — steps away from the idea of having different behaviours by collapsing the differences into one, reducing the disadvantages while preserving most of the advantages. The unified model is based around the updated definitional interpreter presented in the previous chapter. For the new definitional interpreter, the language definition and sequential language definition must be updated.

Definition 5.1.1. *A language L is a structure $\langle P, \Gamma, \gamma_0, O, I \rangle$, with P a set of programs, Γ a set of configurations, $\gamma_0 \in \Gamma$ an initial configuration, O a set of output (values), and I a definitional interpreter that assigns to each program $p \in P$ a function $I_p : \Gamma \to \Gamma_\perp \times O$, where $\Gamma_\perp = \Gamma \cup \{\perp\}$ and O is assumed to form a monoid with identity element ε and binary operation $\bullet$.*

Definition 5.1.2. *A language $L = \langle P, \Gamma, \gamma_0, O, I \rangle$ is sequential if there is an operator $\otimes$ such that for every $p1, p2 \in P$ and $\gamma \in \Gamma$ it holds that $p1 \otimes p2 \in P$ and $I_{p1 \otimes p2}(\gamma) = (\gamma, \varepsilon) \triangleright I_{p_1} \triangleright I_{p_2}$, where $\triangleright : \Gamma \times O \times (\Gamma \to \Gamma_\perp \times O) \to \Gamma \times O$, such that $(\gamma, o) \triangleright f = (\gamma \diamond \gamma', o \bullet o')$, where $(\gamma', o') = f(\gamma)$, and $\gamma \diamond \perp = \gamma$ and $\gamma \diamond \gamma'$ is γ' when $\gamma' \neq \perp$.*

The new sequential definition extends the original definition with propagation of monoidal values through the sequence of programs and handling of erroneous evaluation, such that whenever the execution of a program does not produce a valid configuration, the sequence continues with the last valid configuration. This definition naturally translates to the behaviour seen in many REPLs, where an invalid program does not crashes the REPL, instead the REPL continues with the last valid state.

Using the new definitions, the definition for the unified exploring interpreter model is given.

Definition 5.1.3. *An exploring interpreter for a language $\langle P, \Gamma, \gamma_0, O, I\rangle$ is an algorithm maintaining a current reference (initially r_0), a mapping from references to configurations (initially only mapping r_0 to γ_0), and an execution tree (initially only containing the node labelled r_0) upon which the following actions are executed iteratively:*

- ***execute**(p): let $(\gamma', o) = I_p(\gamma)$, where $p \in P$ is provided as input, and γ is the configuration referenced by the current reference. When $\gamma' \neq \bot$, generate a fresh reference r' such that r' maps to γ', extend the execution tree with the $(r, (p, o), r')$ edge, and transition the current reference to r'.*
- ***revert**(r): let r' be the current reference. When r is an ancestor of r' in the execution tree, change the current reference to r and remove the (unique) path from r to r' without removing r and any nodes and edges used in other paths from r.*
- ***jump**(r): When r is a node in the tree, take r as the current reference.*
- ***display**: provide a structured representation of the exploration tree.*

The unified model makes the usage of references explicit and enforces the representation of the execution environment as a tree — ensuring that every configuration gets a unique reference. The difference between the Stack and Tree behaviours is split into two separate actions: **jump** and **revert**. Jump performs the non-destructive revert — corresponding to the Tree behaviour — and revert is now always destructive — corresponding to the Stack behaviour. Revert also has the requirement that it is only possible to perform a revert to a reference in the current trace such that the current reference is reachable from the reference reverted to. With this requirement and both actions, the Stack and Tree behaviours are both available in the same model.

Splitting the behaviours in two separate actions and ensuring that the revert action always has a destructible path, made it possible to define less destructive semantics for revert compared to the semantics in the Graph behaviour — which destroyed all outgoing paths. Providing a less destructive form, gives interfaces more control to construct complex destructive operations. For example, the Graph destructiveness can be fully described in combinations of **jump** and **revert**.

Ensuring that the execution environment is always represented by a tree, removes the capability of sharing. This provides a more consistent and less complex form when displaying the execution environment, and prevents the introduction of non-existent paths due to cycles. Moreover, the earlier mentioned benefits obtained from sharing are still possible via the new execution environment, because all information is present in the tree. Program conversion can be obtained by placing the references in equivalence classes with respect to their mapped configuration. Execution skipping can be obtained by also storing all programs at the equivalence classes and their resulting configuration. Specifically, a function can transform the exploration tree into a graph, where nodes are equivalence classes and edges still programs.

5.2 Implementation

The implementation stays parameterised, but removes some fields and adds others.

```
data Explorer programs m configs output where
  Explorer :: Language programs m configs output ⇒
    { defInterp :: programs → configs → m (Maybe configs, output)
    , config :: configs   -- Cache the current config
    , currRef :: Ref
    , genRef :: Ref
    , cmap :: IntMap.IntMap configs
    , execEnv :: Gr Ref (programs, output)
    , shadowGraph :: Gr [Ref] (programs, output)
    , configEq :: configs → configs → Bool
    , shadowing :: Bool   -- Shadow the exploration tree in a shadow graph.
    } → Explorer programs m configs output
type Language p m c o = (Eq p, Eq o, Monad m, Monoid o)   -- Using the ContraintKinds extension
type Ref = Int
```

The sharing and backtracking fields are removed, and the *shadowing* field is introduced. Shadowing disables or enables the maintenance of a shadow graph (*shadowGraph*). The shadow graph is implemented as a graph where the references are grouped based on equivalence classes, with these classes being defined by a user supplied function (*configEq*). The shadow graph functions as an optimisation on the earlier described function to obtain the sharing insights and execution skipping optimisation. The remaining fields are identical to those in the earlier implementation. The *execEnv* is still represented as a graph using the fgl library, purely for convenience; it will always follow the requirements of a tree. Furthermore, the definitional interpreter remains executable inside a monad such that effects not captured by the model are possible.

5.2.1 Operations

The operations of the initial implementation must be updated to work with the new *Explorer* type and incorporate the usage of the shadow graph. Furthermore, the new **jump** operation is introduced and the **revert** operation is updated in accordance with the new definition.

The **execute** function is still implemented as an execution of the definitional interpreter on the current configuration with the supplied program.

```
execute :: Language p m c 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 (updateExecEnvs e (p, cfg, o), o)
       Nothing → return (e, o)
```

When the definitional interpreter returns an invalid configuration — indicated by *Nothing* — execute returns the current explorer and any output resulting from the call to the definitional interpreter. In case the definitional produces a valid configuration, the explorer is updated and returned together with any output produced by the definitional interpreter. The explorer is updated by updating the execution environments. The tree is updated by including the edge as detailed in the model definition. The shadow graph is updated only when *shadowing* is enabled by finding the equivalence class to which the configuration referenced by the new reference belongs to. Then, an edge is added to the shadow graph that connects the nodes belonging to the equivalence classes of the new and current configurations, with the executed program and monoidal values as an edge label.

The **revert** action is updated to follow the constraint presented in the model.

```
revert :: Ref → Explorer p m c o → Maybe (Explorer p m c o)
revert r e
  | currRef e ∈ reachNodes =
  jump r e ≫= λe′ → return $ e′ {
    execEnv = mkGraph (zip remainNodes remainNodes) $ cleanEdges reachNodes (labEdges $ execEnv e′)
    , cmap = deleteMap reachNodes (cmap e′)
    , shadowExecEnv = cleanShadowEnv (shadowing e′) reachNodes (shadowExecEnv e′)
  }
  | otherwise = Nothing
```

```
    where
      reachNodes = isUnique (execEnv e) $ reachable r (execEnv e) \\ [r]
      remainNodes = nodes (execEnv e) \\ reachNodes
jump :: Ref → Explorer p m c o → Maybe (Explorer p m c o)
jump r e = case deref e r of
    (Just c) → return $ e {config = c, currRef = r}
    Nothing → Nothing
```

The implementation returns *Nothing* when the reference reverted to is not an ancestor of the current reference or the reference is not in the tree. Otherwise, the path from the reference reverted to containing the current reference, is destroyed while retaining nodes used in other parts and the node of the reference reverted to. When shadowing is enabled, the shadow graph is also updated by removing the nodes removed from the tree.

Furthermore, the revert implementation uses **jump** to update the current reference and current configuration. Similarly to revert, the jump implementation returns *Nothing* when the argument is invalid.

The implementation of most display operations are not affected by the new model, but the *getTraces* function is redundant and therefore removed from the implementation. In addition, to help interfaces display information regarding the program convergence with respect to configuration equivalence classes, the implementation exposes a function that returns all references belonging to the same equivalence class as the argument, and a function that returns all equivalence classes.

```
eqClass :: Ref → Explorer → [Ref]
eqClasses :: Explorer → [[Ref]]
```

5.3 Evaluation

To evaluate the unified model, the implementation is applied to the Idris programming language [36]. Idris is a functional language supporting dependently typed programming. In contrast to the languages used in the earlier evaluation, Idris performs static type checking before a call to **execute**. Furthermore, Idris provides an interactive system via the REPL, with support for proofs, determining call sites, and requesting documentation. As a result, with Idris the exploring interpreter as a component in an interactive environment can be evaluated.

5.3.1 State interpreters

The Idris interpreter is not constructed in the style expected by the exploring interpreter. Instead, the Idris interpreter is described as a monadic computation.

```
idrisMain :: [Opt] → Idris ()
runMain :: Idris () → IO ()
repl = runMain idrisMain []
type Idris = StateT IState (ExceptT Err IO)
```

The interpreter (*idrisMain*) executes in a state transformer monad (*Idris*), which encapsulates the interpreter state (*IState*) — corresponding to a configuration in the language definition — and an exception monad (*ExceptT*) encapsulating an error component (*Err*) and the *IO* monad.

To construct a specialised exploring interpreter for the Idris interpreter, the Idris interpreter must be defined according to the type expected by the exploring interpreter. This can be achieved by defining a new interpreter that encapsulates parts of the existing interpreter.

```
interpreter p s = do
  res ← runExceptT $ execStateT (process "" p) s
  case res of
    (Right s′) → return (Just s′, ())
    (Left err) → putStrLn (show err) ≫ return (Nothing, ())
```

The encapsulating interpreter performs an *Idris* computation with the second argument (s) as the starting state for the computation. The result of the computation is extracted and wrapped according to the expected result from the exploring interpreter.

With the encapsulating interpreter, the Idris language is fully defined to utilise the exploring interpreter implementation and an exploratory REPL for Idris can be defined.

```
repl = do
setup ← runExceptT $ execStateT initREPL idrisInit
initialState ← case setup of
  (Left err) → error (show err)
  (Right state) → return state
let e = mkExplorer True (≡) interpreter initialState
loop e
where
  loop e = do
    putStr ("Exploring Idris(curr state: " ++ (show $ currRef e) ++ ")> ")
    input ← getLine
    case break isSpace input of
        -- Other meta-commands, like revert and jump.
```

```
(":inspect", mint) | Just ref_id ← readMaybe (dropWhile isSpace mint)
      → handle_inspect ref_id e ≫ loop e
      | otherwise → putStrLn "Inspect requires an integer argument" ≫ loop e
(remain, _) → let res = parseCmd (config e) "input" input in
   case res of
     (Right (Right cmd)) → Language.Explorer.Monadic.execute cmd e ≫= loop ∘ fst
     (Right (Left err)) → putStrLn (show err) ≫ loop e
     (Left err) → putStrLn (show $ messageText err) ≫ loop e
```

5.3.2 Shadow graph

Maintaining the shadow graph as part of the implementation is not fully required to obtain its benefits. As stated earlier, it is possible with the display functions to construct equivalence classes without having to maintain a shadow graph in the implementation.

Table 5.1 compares the usage of the shadow graph to an implementation that calculates the equivalence classes every time. The table shows that calculating the equivalence classes every time becomes a bottle-neck during the exploration. When the distribution is high — meaning many equivalence classes — the average waiting time is almost half a second.

The evolution of the time required with a probability of 0.75 is displayed in Figure 5.1. The figure shows that the non-cached version already grows quadratic in contrast to the linear growth seen with the cached version.

With the non-cached version, requesting information regarding equivalence classes frequently results in a noticeable delay for the user, which grows with the number of configurations and the number of equivalence classes. By caching the equivalence classes and updating this information after every execute operation, this delay is significantly reduced and the information is readily available after an execute operation.

Purely caching the equivalence classes does not constitute including it in the implementation. However, when combined with the option of the explorer to utilise information from the shadow graph, for example to perform execution skipping, and to be able to provide the same features present in the earlier implementation, the inclusion is beneficial.

5.3.3 Interactive environments and the exploring interpreter

When the exploring interpreter is part of a larger interactive environment, the other components can be combined with the exploration session to increase the exploratory experience and enhance the functionality of other components. In Idris, there are several meta-commands that can be used in such a manner. Table 5.2 gives an overview of some of these commands with a small description describing the functionality of the command. In

Table 5.1: Average time to request equivalence classes by executing 100 programs. Results are the average of 10 runs of the experiment, compiled with ghc-8.0.2. Probability denotes the probability a program results in a new equivalence class. With a 0 probability, all configurations are grouped in the same equivalence class and with a probability of 1, every configuration is placed in separate equivalence class. Cached eq-classes are requested via the *eqClasses* function exported by exploring interpreter. The non-cached eq-classes are obtained via a function that transforms the tree into a list of equivalence classes.

Probability for new equivalence class	Average time for cached eq-classes (ms)	Average time for non-cached eq-classes (ms)
0.0	2.0646	45.1253
0.25	14.2333	129.3170
0.5	14.5434	254.5370
0.75	14.9909	324.8710
1	15.2111	421.7530

this section, the interaction of the commands *search*, *whocalls*, and *elab* with the exploring interpreter are discussed.

The *search* command searches through a configuration to find all values matching the supplied type. For example, the plus operator can be searched by executing the command: *:search Nat* $\rightarrow$ *Nat* $\rightarrow$ *Nat*. By utilising the exploring interpreter, a search can be executed over multiple configurations at once. This allows a user to perform structured searching of the exploration tree. Furthermore, the result of a search can be used to modify the exploration tree. For example, when a term is found in a different configuration than the current one, the term can be used as the argument to an execute operation, injecting the term in the current exploratory trace. This operation can be performed by the interactive environment, making it trivial for a user to update the existing trace with information from another trace. Another use-case is switching to an other trace based on the searched information, where the search results can be used to select the configuration to jump to.

The *whocalls* command shows all callers of a function. A user can use this command in combination with the exploring interpreter to modify the exploration tree. For example, by deleting all traces without callers to a specific function or by updating all traces with callers to a function with an updated definition. Again, modifying the exploration tree can be performed by the interactive environment, and by utilising other components in the environment, a more feature full exploration can be performed.

The last command discussed is the *elab* command. This command starts the interactive

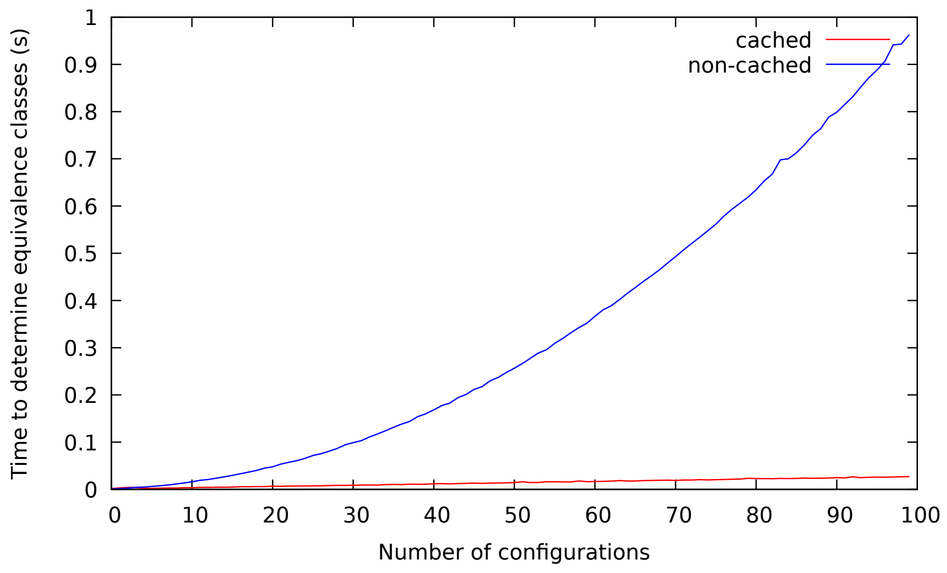


Figure 5.1: Evolution of time to determine equivalence classes with the cached and non-cached approaches, using a probability of 0.75 to create new equivalence classes. Averaged over 10 runs.

prover to prove the supplied hole. The interactive prover is a separate component and operates on a new language, specific to proving. By integrating the proving system with the exploring interpreter, it is possible to start the proving system on a chosen configuration from the exploratory session, create a proof, and inject the proof as a program into the exploratory session. This way, the user does not need to end the exploratory session when wanting to prove a hole, and the proof can immediately be utilised in the current exploratory session, while retaining the usage of the original proving system.

The three discussed commands all operate on a configuration, which makes integration with the exploring interpreter trivial. Furthermore, debugging is not present in Idris, nonetheless, the *elab* command provides a similar experience and can be substituted for a debug command in an environment supporting debugging. Thus, by placing the exploring interpreter as a component in an interactive environment, existing components can be re-used and integrated with the exploring interpreter to increase the exploratory experience.

Table 5.2: Some of the meta-commands available in the Idris REPL

Command name	Description
search	Search a configuration for a value of the supplied type
apropos	Search a configuration for the given name
browse	Give information about a namespace
whocalls	List the names of the callers for the supplied name
callswho	List the names of the callees for the supplied name
doc	Show documentation of the supplied name
metavars	Show remaining proof obligations for supplied meta variable
total	Check if the supplied name is total
showproof	Show the proof for the supplied name
elab	Start the elaboration shell to prove a meta variable

6

Language Parametric Interfaces for Exploratory Programming

In this chapter, language parametric interfaces for exploratory programming are introduced. Language parametric interfaces are re-usable interfaces that are made concrete by a language. Via language parametric interfaces, common boilerplate shared between interfaces is abstracted into re-usable components, while retaining flexibility for language specific functionality.

I make a distinction between two type of interfaces: embedded interface and non-embedded interfaces. Embedded interfaces are developed in the same language as the exploring interpreter. As a result, an interface can easily communicate with the exploring interpreter via function calls. This makes the implementation of an embedded interface simpler, but it also restricts the applicability of the interface, because it can only be used on languages developed in the language used by the interface and exploring interpreter. Examples of embedded interfaces are the REPL for *While* and the REPL for Idris. Nonetheless, embedded interfaces are not restricted to purely REPLs, since it is possible to construct an embedded GUI interface.

Non-embedded interfaces are developed in a language different than the language used in the development of the exploring interpreter. This prevents an interface to communicate with the exploring interpreter via simple function calls. Instead, a communication format is required to enable communication between the interface and the exploring interpreter. Hence, interface implementation is more complex, because a data communication format and implementation is required on both the interface side and the exploring interpreter side. Nonetheless, it makes the interface applicable to any exploring interpreter following the communication format used, allowing more re-usability. An example of a non-embedded interface is the eFLINT web-interface.

6.1 Embedded interfaces

The *While* REPL, introduced in chapter 4, and the Idris REPL, introduced in chapter 5, share many functionalities: managing an exploring interpreter, parsing and execution of meta-commands, parsing programs and executing the parsed programs, parsing exploring interpreter commands, displaying prompts, and handling output produced by an execution. Providing the REPL loop, managing the exploring interpreter, and handling exploring interpreter commands are language independent. Furthermore, the other functionalities are language parametric and can be abstracted into a language parametric REPL.

The language parametric REPL is parameterised by the prompt of the REPL, the parser of the language, the meta-command prefix, the handler for meta-commands, the table for exploring interpreter commands, and the output handler.

```
repl :: (Eq p, Eq o, Monoid o, MonadIO m) ⇒ Repl p m c o
repl prompt parser metaPrefix metaHandler commandTable outputHandler ex = do
  minput ← liftIO ∘ Rl.readline ∘ prompt $ ex
  case minput of
    (Just input) → do
      liftIO $ Rl.addHistory input
      if metaPrefix ‘isPrefixOf‘ input then runMeta input else runExec input
    Nothing → return ()
  where
    repl′ = Main.repl prompt parser metaPrefix metaHandler commandTable outputHandler
    runMeta input =
      let (pcmd, args) = break isSpace input in
        case find (λ(cmd, _) → (metaPrefix ++ cmd) ≡ pcmd) metaTable of
          Just (_, f) → f args ex ≫= repl′
          Nothing → metaHandler input ex ≫= repl′
    runExec input =
      case parser input (config ex) of
        (Just program) → L.execute program ex ≫= λ(newEx, out) → (outputHandler out ≫ repl′ newEx)
        Nothing → repl′ ex
type Repl p m c o = Prompt p m c o → Parser p c → String → CommandTable p m c o →
  MetaHandler p m c o → OutputHandler m o → Explorer p m c o → m ()
type Prompt p m c o = Explorer p m c o → String
type Parser p c = String → c → Maybe p
type CommandTable p m c o = [(String, String → Explorer p m c o → m (Explorer p m c o))]
type MetaHandler p m c o = String → Explorer p m c o → m (Explorer p m c o)
type OutputHandler m o = o → m ()
```

The parameterisation of the prompt allows a REPL to display custom information to the user. The meta-command parameters allow injection of the exploring interpreter commands in the namespace of the meta-commands, and the meta handler is used to dispatch to when a meta-command is not an exploring interpreter command. Exploring interpreter commands are given to the REPL in the form of a command table. By making the command table a parameter, a language can modify the syntax of the exploring interpreter commands, for example when a collision arises. Furthermore, it is possible to combine independently developed commands into one command-table. The distinction between the meta-handler and the command table is not necessary, because the meta-handler is typed identical to the handlers in the command table. Therefore, a language can inject the meta-commands into the command table without the need for the meta-handler. However, by providing the option of a meta-handler parameter, existing functionality of a language can be utilised, as is the case with Idris, and a clear distinction between language generic and language specific operations is made. Furthermore, the meta-handler functions as a catch-all clause for when a command is not in the command-table, allowing an implementation to handle the passing of a missing meta-commands language specifically.

With the language parametric REPL, the Idris REPL can be defined as follows.

```
idrisRepl = do
  setup ← runExceptT $ execStateT initREPL idrisInit
  initialState ← case setup of
    (Left err) → error (show err)
    (Right state) → return state
  let e = mkExplorer True (≡) interpreter initialState
  repl (const "Idris> ") (flip simpleParser) ":" metaTable metaHandler (\_ → return ()) e
```

The Idris REPL only performs initialisation, by constructing the initial state and an explorer. The remainder of the REPL functionality is obtained by calling the language parametric REPL — identified by the *repl* function.

6.2 Non-embedded interfaces

Non-embedded interfaces require a communication format to relay information between the interface and the exploring interpreter. When there is no standard format, the classical n times m problem arises, where there are n interfaces and m exploring interpreter servers, and the exploring interpreter servers all have their own communication format. This requires an interface to implement m formats to support a wide range of exploring interpreter implementations. This work must be repeated by n interfaces, resulting in n

times m implementations. The n times m problem is classical since it also occurs with compilers and with text editors supporting programming with IDE like features. In compilers, the problem occurs when there are n languages compiling to m platforms. In editors it occurs because there are n editors supporting m languages, requiring every editor to implement IDE features m times.

For compilers, the problem is solved by systems like Clang [37] and GCC [38], which abstract the m targets by providing one common language. As a result, a compiler only has to generate the common language, from which the system can generate multiple target languages. For editors, the problem is reduced by the Language Server Protocol (LSP) and the Debug Adapter Protocol (DAP). These protocols describe a protocol for IDE functionality, which allows an editor to connect to a server that implements the protocol. As a result, an editor only has to implement the protocol and can support languages that have a server implementing the protocol for free, allowing different editors to share the same server implementation.

Both solutions rely on a common data format to solve the n times m problem. To solve this for exploring interpreters, I introduce the exploring interpreter protocol (EIP). The protocol functions as an interface between the front-end or GUI of an environment and an exploring interpreter server, removing the fixed dependency from the front-end on the back-end, i.e. it is trivial to swap the back-end by an other back-end while using the same interface, and the other way around.

The protocol is as an instance of the JSON RPC 2.0 protocol[1], and is described using TypeScript interfaces similarly as LSP and DAP. The JSON RPC 2.0 protocol defines a request object, response object, and an error object, which are all encoded as JSON objects. A request object contains an identifier, a method name and the type capturing the parameter(s) of the method (if any). A response object contains an identifier for the request to which it responds and either a result or an error object. The result can be any encoded JSON object and the error object contains a unique error code, a short descriptive error message, and optional extra error data that can be any object.

The requests and response pairs of the protocol contain the operations corresponding to the actions of the exploring interpreter algorithm, of which we detail the specification **jump**, **revert**, and **execute**, and operations for accessing (parts of) the exploring interpreter's execution tree (i.e. variants of **display**), such as getting all leaves of the tree or the current trace. The full list of methods is given in Table 6.1, and the full protocol specification is available in Appendix A.

[1]`https://www.jsonrpc.org/specification`

Table 6.1: The methods provided by the exploring interpreter protocol.

Command	Description
execute	see **execute** in Definition 5.1.3 and the protocol specification in section 6.2
revert	see **revert** in definition 5.1.3 and the protocol specification in section 6.2
jump	see **jump** in definition 5.1.3 and the protocol specification in section 6.2
getCurrentReference	gets the reference labelling the current node
getAllReferences	returns all references used as a label
getRoot	returns the reference labelling the root node
deref	gets the configuration assigned to the given reference
getExecutionTree	gets the execution tree in the form of the current node a list of edges and list of nodes
getTrace	gets the edges representing the path from the root node to the current node
getPath	gets the edges representing that path between the nodes labelled by two given references
getLeaves	gets the references labelling the nodes without outgoing edges
metaCommand	execute a meta-command via the meta handler of the language

A **jump** operation must be encoded as a request with the method specified as “jump” and the parameter an object containing a reference.

```
interface JumpRequest extends RequestMessage {
    method: "jump";
    params: JumpParams;
}
interface JumpParams {
    reference: uinteger;
}
```

As a result, a jump specific response is given which has a null result or a jump error.

```
interface JumpResponse extends ResponseMessage {
    result?: null;
    error?: JumpError;
}
```

```
interface JumpError extends ResponseError {
    code: DefaultErrorCodes | ReferenceNotInTree;
}
```

A null result indicates a successful jump and the reference present in the request is now the current reference. An error object can either encode one of the default errors of JSON RPC, e.g. “method not found”, or a “reference not in tree” error.

A **revert** operation is encoded similarly as a jump request, but using method name “revert” and a revert parameter instead of a jump parameter [1].

```
interface RevertRequest extends RequestMessage {
    method: "revert";
    params: RevertParams;
}

interface RevertParams {
    reference: uinteger;
}
```

The response of a **revert** request contains the deleted nodes, indicated by references.

```
interface RevertResponse extends ResponseMessage {
    result?: [uinteger];
    error?: RevertError;
}

interface RevertError extends ResponseError {
    code: DefaultErrorCodes | ReferenceNotInTree | ReferenceRevertInvalid;
}
```

A revert error is like a jump error but has an additional alternative for the case the given reference is not an ancestor of the current reference.

The **execute** operation has a request with the method specified as “execute” and the parameter an execute parameter object. The execute parameter object contains a string representing the program to execute.

```
interface ExecuteRequest extends RequestMessage {
    method: "execute";
    params: ExecuteParams;
}

interface ExecuteParams {
    program: string;
}
```

[1] Revert and jump parameters are defined separately to simplify future extensions.

As a response, the **execute** operation gives an execute result or an execute error. An execute result is an object containing the fresh reference and any output produced by the program. The program error contains a default error or an error indicating the supplied program can not be parsed, or an error indicating the execution of the program. In the last case, the output of the execution is placed in the data field of the error object.

```
interface ExecuteResponse extends ResponseMessage {
    result?: ExecuteResult;
    error?: ExecuteError;
}

interface ExecuteResult {
    reference: uinteger;
    output: string;
}

interface ExecuteError extends ResponseError {
    code: DefaultErrorCodes | ProgramParseError | ExecuteError;
}
```

6.2.1 Implementation

As part of the protocol, I propose an architecture using websockets, of which an overview is given in Figure 6.1. The proposed architecture contains a front-end and a back-end. The front-end handles the connection from the interface side, which is mostly focused on the creation of requests and the handling of responses. The back-end handles the connection from the exploring interpreter side, which is focused on the handling of requests and creation of responses.

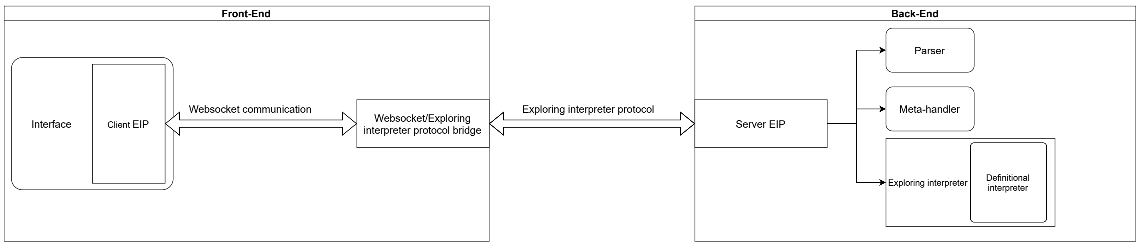


Figure 6.1: Overview of the client-server architecture implemented on top of the exploring interpreter protocol. The rectangles with rounded corners are language dependent and the other rectangles are language generic.

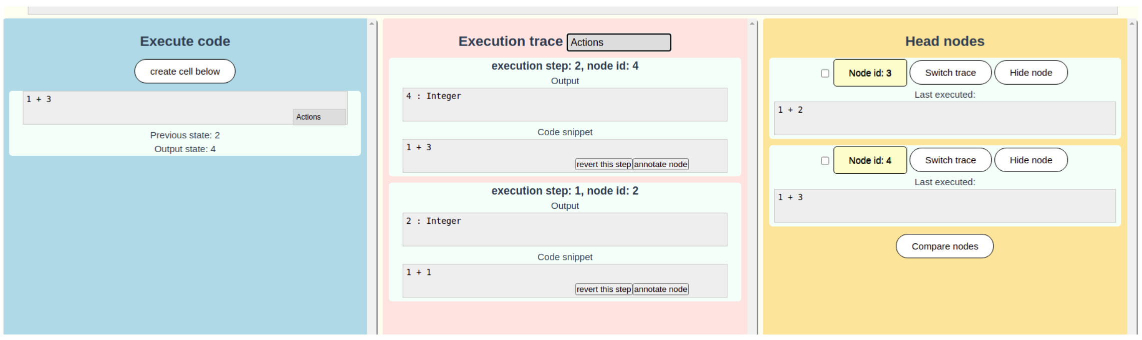
The front-end contains an interface implementing the abstract client EIP. The client EIP, partly shown in Figure 6.2, handles the communication with a bridge. The communication is performed over web-sockets, which provides real-time interaction with the interface and allows the client EIP to send bare JSON over the network. The bridge transforms the JSON into a correct EIP request and sends it over a socket to the server. The bridge also handles the response from the server and extracts the relevant information for the interface and sends it to the interface over the websocket. Since the exploring interpreter protocol is much simpler than HTTP, it does not use HTTP. Hence, the bridge must ensure that all the bytes of a response are received from the server before forwarding it to the interface. The client EIP is abstract because the *on* methods determine how an interface responds to responses of the EIP server, and are language dependent.

The back-end contains a server EIP that opens a socket and receives messages from the bridge. The server EIP ensures that all the bytes of an EIP request are received from the bridge and translates a request to functions on the parser, meta-handler, and exploring interpreter. Furthermore, the EIP server translates responses of these function calls into valid EIP responses and sends the response back to the bridge.

With the client EIP and the server EIP, the core part of the architecture is abstracted away and language generic. On the front-end, an interface only needs to implement the *on* methods to implement how responses affect the interface. On the back-end, an implementation only needs to fulfil Definition 5.1.1, have a parser and a meta-handler, and the remaining parts of the back-end are automatically obtained. These abstractions were utilised by the author of [39] to adapt the interface from the corresponding paper — not running on the exploring interpreter protocol — to support Idris without modifications on the Idris back-end. In total, the implementation took around 100 extra lines of code, which were mostly located in the *on* methods. The resulting interface is shown in Figure 6.3.

```
export abstract class EIP {
    socket: Socket;
    requests: Map<string, RequestMessage> = new Map();

    constructor(socket: Socket) {
        this.socket = socket
        socket.on("data", (data) => {
            this.handleResponse(data);
        });
        console.log(this);
    }

    ...

    handleResponse(resp: any) {
        let r = JSON.parse(resp) as eip.ResponseMessage;
        // Handle possible error cases
        ...

        switch(req.method) {
            case "execute":
                this.onExecute(req as eip.ExecuteRequest,
                               r as eip.ExecuteResponse);
                break;
            ...
        }
    }

    execute(params: eip.ExecuteParams) {
        this.send(new ExecuteRequest(params));
    }

    abstract onExecute(req: eip.ExecuteRequest,
                      resp: eip.ExecuteResponse): void

    abstract onUnkownError(resp: eip.ResponseMessage): void;
    abstract onError(req: eip.RequestMessage,
                    resp: eip.ResponseMessage): void;
}
```

Figure 6.2: Part of the abstract client exploring interpreter protocol class, which abstracts away all the protocol related implementation details.



```
1 + 3
```

Previous state: 2
Output state: 4

```
4 : Integer
```

```
1 + 3
```

```
2 : Integer
```

```
1 + 1
```

```
1 + 2
```

```
1 + 3
```

Figure 6.3: An exploratory programming interface using the exploring interpreter protocol and implementation.

7

Exploratory Polyglot REPLs

Polyglot REPLs enable a programmer to mix languages in the same REPL session. For example, a construction of *While* mixed with a *Lambda* language enables a programmer to use functions which are not present in the *While* language. With a fine-grained polyglot REPL supporting *While* and *Lambda*, the following session is possible.

```
Lang> start = 0
Lang> end = 100
Lang> add = lambda l lambda r l + r done done
Lang> addWithS = add(0)
Lang> while (start =< end) print(start);
  start = addWithS(start + 1); addWithS = add(start) done
0
1
3
7
15
31
63
```

The example first defines two variables, *start* and *end*, which are language constructs from *While*. Then, the *add* function is defined. The definition of the *add* function uses the assignment statement from *While* in combination with the lambda from *Lambda*. Furthermore, inside the lambda, addition is used which is a language construct from *While*. Next, the *add* function is applied once to create a function, *addWithS*, that adds 0 to the provided argument. Then, a while loop is constructed with a body that first displays the value of the *start* variable, then assigns the *start* variable the value of calling the *addWithS* function with the *start* value increased by one as the argument, and finally updates the *addWithS* function to store the updated value of the *start* variable.

In the example, language constructs from *While* and *Lambda* are freely mixed, enabling a much more powerful form of programming than possible when using the languages in isolation. Furthermore, it must be emphasised that the languages are independently developed and it is trivial to obtain a REPL supporting only *While* or *Lambda*, or a REPL with *Lambda* or *While* mixed with a completely different language.

In this chapter, I outline the method for constructing the example polyglot REPL, starting with the introduction of the running example. After introducing the running example, the construction of a trivial REPL that understands both languages but is not a valid polyglot REPL, to highlight the difficulty of polyglot REPLs, is shown. Following, a solution to the presented problem is introduced, starting with a focus on abstract syntax that is slowly developed into a full example with concrete and abstract syntax, and semantics. Finally, an extension on the method is presented to make the method less prone to incompatible polyglot REPLs.

7.1 Running example

Our running example consists of the *While* language, introduced in chapter 4, and the *Lambda* language. *Lambda* is an implementation of untyped lambda calculus and is encoded as follows.

```
data Expr = Var String | Abstraction String Expr | Application Expr Expr
lambdaInterpreter :: Expr → Config → Config
data Config = Config {env :: Env}
type Env = Map.Map String Expr
initialConfig = Env {env = empty}
```

The *Expr*, *lambdaInterpreter*, *Config*, and *EmptyConfig* form a language according to Definition 2.2.1[1] and the interpreter is implemented according to Definition 2.2.2, making the language sequential.

7.2 Polgot REPLs via language composition

A polyglot REPLs allows submission of code snippets from multiple languages, and executes these snippets in the same context, allowing interoperability between the used languages and in a fine-grained REPL, mixing of language constructs.

[1]The original definitions are used for simplicity, since the interpreter is abstracted away at a later point.

A polyglot REPL can also be seen as an ordinary REPL for a polyglot language — a language consisting of multiple languages. This way, polyglot REPLs can be obtained via language composition. In our example, *While* and *Lambda* can be composed into a new language as follows.

```
whileLambdaInterpreter = WhileLambdaProgram → Config → Config
whileLambdaInterpreter (While p) (cw, cl) = (whileInterpreter p cw, cl)
whileLambdaInterpreter (Lambda p) (cw, cl) = (cw, lambdaInterpreter p cl)
data WhileLambdaProgram = While While.Command | Lambda Lambda.Expr
type Config = (While.Config, Lambda.Config)
initialConfig = (While.initialConfig, Lambda.initialConfig)
```

To accept both programs, the new language keeps track of both language configurations and implements the interpreter by dispatching to the language specific interpreter. A REPL for this language is fully functional and accepts programs of both languages. However, there is no interoperability between the languages since the languages still operate in their own context. But polyglot programming uses multiple languages in the same context. Thus, simple language composition does not result in valid polyglot REPLs, because a valid polyglot REPL requires combining the context of different languages.

7.3 Combining context

There are different methods for combining the context of different languages [25, 26], but all depend on a common data format to which the languages can translate their context to or can operate in.

In the presented approach, funcons will function as the common data format to which languages translate their semantics, enabling operating the language with the same interpreter and the same context. Funcons enable specification of different paradigms, which enables composition of languages across paradigms. Furthermore, funcon definitions are immutable and specify the semantics of the language. This makes composition easier, because incompatible funcon versions between languages is not possible, and a language only has to translate its abstract syntax into funcon terms with all semantics handled by funcons. Furthermore, the translation to funcons is done at a very high level of abstraction, decreasing the effort needed for a translation.

With funcons as the common data format, languages only need to translate their abstract syntax into funcon terms. For this, we introduce a type class defining the interface for the translation.

```
class ToFuncons programs where
  toFuncons :: programs → Funcons
```

With our common data format and the type class to distinguish between composable languages, our composition of *While* and *Lambda* can be written as follows.

```
whileLambdaInterpreter :: WhileLambdaProgram → Config → Config
whileLambdaInterpreter p cfg = defInterpreter (toFuncons p) cfg
defInterpreter :: Funcons → Config → IO Config
instance ToFuncons WhileLambdaProgram where
  toFuncons (While p) = toFuncons p
  toFuncons (Lambda p) = toFuncons p
```

The composition now uses the funcons interpreter to evaluate the languages, with the configurations being funcon specific and shared between the different languages. As a result of the composition based on the common data format, values from *While* and *Lambda* can be used across the languages.

However, the current composition is solely focused on *While* and *Lambda*, but we might want to compose another language to extend our polyglot REPL further. When introducing a new language, the allowed programs in our composition must be updated because the programs in our compositions are explicit, requiring new implementation of the *ToFuncons* instance. Furthermore, the current polyglot REPL is coarse, we can use *While* and *Lambda* programs in the same context, but it is not possible to freely mix the language constructs of both languages, because only their translation to funcons is shared and not their abstract syntax.

7.4 Signature composition

To overcome explicit program notation, which requires an instance definition and prevents fine grained composition, data types á la carte is integrated as the method in which languages describe their signature. Via this approach, languages can be composed using the co-product of their signatures, which also enables finer grained composition than just top-level composition, allowing fine grained polyglot REPLs.

With the new approach, *While* is encoded as follows.

```
data Literal a = LitBool Bool | LitInt Int deriving Functor
data Expr a = Leq a a | Plus a a | Id String deriving Functor
data Command a = Seq a a | Assign String a | Print a | While a a a | Done deriving Functor
type Sig = Literal :+: Expr :+: Command
```

And *Lambda* is encoded similarly.

```
data Expr a = Var String | Abstraction String a | Application a a deriving Functor
type Sig = Expr
```

Both languages export their signature (*Sig*), which is used in the composition.

```
type LambdaWhile = Lambda.Sig :+: While.Sig
```

Note that the type *LambdaWhile* is just for clarity and is not required. Furthermore, we are free to extend the language with other signatures. This new representation requires an update to the *toFuncons* instance, which must now be a valid algebra to be used in a catamorphism over the signatures to translate signatures into funcon terms.

```
class ToFuncons f Funcons where
  tofuncons :: f Funcons → Funcons
 $ (derive [liftSum] ['' ToFuncons])
interpreter p cfg = def_interpreter (cata toFuncons p) cfg
```

Valid programs in the polyglot REPL are now of type *Term* (*Lambda.Sig* :+: *While.Sig*), which intertwines the abstract syntax of the languages, allowing usage of *Lambda* expression inside *While* construct, and the other way around.

7.5 Concrete languages

With a solution for abstract syntax and semantics, only concrete syntax remains. To extend the current approach with support for concrete syntax, generalised parsing [40] is used. Generalised parsing enables parsing of all context free languages and computes all derivations of input. Computing all derivations of input is crucial when perform fine grained composition in polyglot REPLs, because parser composition of arbitrary languages can result in ambiguities.

With the parsing library, a parser for the *Lambda* language can be defined as follows.

```
pLambda :: Expr :<: f ⇒ BNF Token (Term f)
pLambda = "Lambda"
  ⟨::=⟩ iVar ⟨$$⟩ id_lit
  ⟨||⟩ iAbstraction ⟨$$⟩ (keyword "lambda" **⟩ id_lit) ⟨**⟩ (pLambda ⟨** keyword "done")
  ⟨||⟩ iApplication ⟨$$⟩ pLambda ⟨**⟩ (keychar '(' **⟩ pLambda ⟨** keychar ')')
```

The left hand side of the ⟨::=⟩ operator names the production of this parser and the right hand side describes the rules for the production. Furthermore, the parsing result is defined open by stating that it returns a parser of type *Term f*, which is valid as long as the *Expr* type is subsumed by *f* — indicated with the *Expr* :<: *f* type constraint. As a result, composition with other parser is possible, as long as the final signature subsumes the *Expr* type.

With the parser constraints, a language is completely specified by its parser, signature, and its instance definition of the *ToFuncons* type class. This enables language composition by compositions of parsers, which is possible with the choice($\langle || \rangle$) operator.

$$pWhileLambda = \texttt{"LambdaWhile"}\ \langle ::= \rangle\ pWhile\ \langle || \rangle\ pLambda$$

The choice operator first tries the parser of the left side. If that parser does not parse the input, the parser on the right side is tried. This assigns a higher priority to the parser on the right side, thus if a program is parsable by both parsers, it will be parsed by the first and thus translated to funcons by the first language. Nonetheless, it makes freely parsing two languages in the same REPL possible.

7.6 Exploratory polyglots

With the addition of concrete syntax, most requirements are met to utilise the exploring interpreter from chapter 5 and the language parametric REPL from chapter 6.

To fully meet the requirements and obtain sequential languages without requiring modifications for different compositions, a generalisation of the *whileLambdaInterpreter* is defined.

$$\begin{aligned}&interpreter :: Term\ f \rightarrow Config \rightarrow IO\ (Maybe\ Config, ())\\&interpreter\ term\ cfg = def\_interpreter\ (cata\ toFuncons\ p)\ cfg \gg\!= \lambda cfg \rightarrow return\ (Just\ cfg, ())\end{aligned}$$

With the generalisation, wrapping the result configuration in a *Maybe* monad, and taking () as the monoid, our languages follow Definition 5.1.1.

Using the approach outlined in [9], the composed language can be made sequential by extending the language with a sequence operator and implementing the interpreter for the extended language according to Definition 5.1.2.

For the sequence operator, a parser is defined that takes the parser of the composed language and extends this with support for sequencing using the supplied sequence operator.

$$\begin{aligned}&pSequential :: String \rightarrow Parser\ f \rightarrow BNF\ Token\ (Seq\ (Term\ f))\\&pSequential\ seqOp\ pmain = \texttt{"programs"}\ \langle ::= \rangle\ Program\ \langle \$\$ \rangle\ pmain\\&\quad \langle || \rangle\ SeqProgram\ \langle \$\$ \rangle\ (pmain\ \langle ** \ keyword\ seqOp)\ \langle ** \rangle\ pmain\\&\mathbf{data}\ Seq\ f = Program\ f \mid SeqProgram\ f\ f\end{aligned}$$

The interpreter for the sequenced language is defined as follows.

```
seqInterpreter :: Seq (Term f) → Config → IO (Maybe Config, ())
seqInterpreter (Program p) cfg = interpreter p cfg
seqInterpreter (SeqProgram p1 p2) cfg =
  interpreter p1 cfg ≫= λ(mcfg, mval) → interpreter p2 (fromMaybe cfg mcfg) ≫= return ∘ second (mval‘mappend‘)
```

Using the combination of the *seqInterpreter* and *pSequential*, language compositions become sequential languages without needing sequentiality from the composition itself.

Now, both the exploring interpreter and the REPL can be used for any composition of languages, enabling the creation of free exploratory polyglot REPLs via language composition.

```
whileLambdaRepl = polyglotREPL WhileLambda.lexerSettings (pWhileLambda :: Parser (WhileLambda.Sig))
polyglotREPL lexerSettings parser = do
  (cfg, runopt) ← setup []
  let ex = mkExplorer False (\_ → \_ → False) (seqInterpreter (def_interpreter runopt)) cfg
  let replParser = λp → \_ → listToMaybe $ parse (pSequential ";" parser) (lexer lexerSettings p)
  repl (\_ → "Lang> ") replParser ":" metaTable (\_ → λex → return ex) (\_ → return ()) ex
```

The polyglot REPL first performs setup required by the funcons definitional interpreter. It then constructs the *replParser* which parses input using the sequential parser and takes the head of the results as the parse result, if any. Finally, the polyglot REPL calls the language parametric REPL with default values.

7.7 Fine grained polyglots

The creation of polyglot REPLs is now possible, but the created REPL only supports coarse grained polyglot programming. Hence, the example session introduced at the beginning of this chapter is not yet obtained.

To allow fine-grained polyglot programming, parsers are defined as higher order parsers — parser taking parsers as arguments and returning new parsers. With higher order parser, the languages can be defined as open, allowing other languages to be embedded into the open languages. For example, the parser for *While* commands can be defines as a higher order parser as follows.

```
pWhile :: Command :<: f ⇒ Parser f → Parser f → Parser f → Parser f
pWhile pmain pexpr pcond = mkRule $ iAssign ⟨$$⟩ id_lit ⟨**⟩ (keychar '=' **⟩ pexpr)
  ⟨||⟩ iPrint ⟨$$⟩ (keyword "print" **⟩ (keychar '(' **⟩ pexpr ⟨** keychar ')'))
  ⟨||⟩ (λe → λc → iWhile e c c) ⟨$$⟩
    (keyword "while" **⟩ (keychar '(' **⟩ pcond ⟨** keychar ')')) ⟨**⟩ pmain ⟨** keyword "done"
  ⟨||⟩ pSeq pmain
type Parser f = BNF Token (Term f)
```

The parser takes as argument the parser for statements in while bodies, the parser for expressions in statements, and the parser for conditionals. With this open definition, any parser can be injected into the *While* language, because all parsers result in a *Term f*, which is open due to the used signature encoding. Thus, combining *While* expressions and *Lambda* expression can be done as follows.

```
pWhileLambda :: (L.Expr :<: f, W.Literal :<: f, W.Expr :<: f, W.Command :<: f) ⇒ Parser f
pWhileLambda = "WhileLambda"
  ⟨::=⟩ L.pLambda pWhileLambdaExpr
  ⟨||⟩ W.pWhile pWhileLambda pLambdaWhileExpr pLambdaWhileExpr
pLambdaWhileExpr :: (L.Expr :<: f, W.Command :<: f, W.Expr :<: f, W.Literal :<: f) ⇒ Parser f
pLambdaWhileExpr = "LWexpr"
  ⟨::=⟩ L.pLambda pWhileLambda ⟨||⟩ W.pExpr pLambdaWhileExpr
```

This composition of parsers results in a language that allows usage of *Lambda* expressions inside *While* statements, and *While* expressions inside *Lambda* expressions. Via this embedding, the example polyglot REPL session introduced at the beginning of this chapter is possible, and a method is available for the construction of exploratory polyglot environments via language composition.

7.8 Constrained compositions

One disadvantage of the usage of open languages, is that it is easier to construct combinations that translate to erroneous funcons. For example, with a certain combination of parser it is possible to have a while loop on the left hand side of an assign statement, which results in a invalid funcon translation.

However, the current parsers all have type *Term f* as their parse result type and there is no check if the constructed parser always results in an invalid funcon translation. As a result, the user combining the languages must pay close to attention to the semantics of the language, which is not ideal and error prone.

Instead, it would be beneficial if a parser can put a constraint on the parsers it accepts. For example, stating that a parser only accepts parser that parse signatures that translate to funcons that return a value on evaluation. For example, constraining the parser passed to the *Lambda* parser to be an expression, where an expression is defined as a funcon that returns a value on evaluation.

```
pLambda :: Functor f, Expr :<: f, IsExpr f ⇒ Parser f → Parser f
```

In the example, the functor f is now required to be an instance of the *IsExpr* type class, where the *IsExpr* type class is an empty type class. However, both parser are still of type

Parser f, which means that the resulting parser also must be an instance of type *IsExpr*. Since all parsers are of the same type, all types must be an instance of the *IsExpr* class to use the *Lambda*. This requirement makes it impossible to compose *Lambda* with *While*, because *While* commands do not return a value, therefore, are not part of the *IsExpr* type class.

Thus, having one parser type is insufficient when wanting to constrain parser used in open languages. Instead, it should be possible to combine different parser into new parsers such that the combined parser contains both the parsable terms from the first parser and the parsable terms form the second.

$$\mathit{pLambda} :: \mathit{Functor}\ f, \mathit{Expr} :<: (f :+: \mathit{Expr}), \mathit{IsExpr}\ f \Rightarrow \mathit{Parser}\ f \rightarrow \mathit{Parser}\ (f :+: \mathit{Expr})$$

Now, the *Lambda* parser takes an argument of type *Parser f* but returns a parser of type *Parser* $(f :+: \mathit{Expr})$.

However, now it is impossible to combine the *Lambda* parser with parsers already containing the *Expr* type, because the resulting type of the parser is then an ambiguous composition, making injection impossible. For example, if our parser argument is of type *Parser* $(g :+: \mathit{Expr})$, the type of the resulting parser of passing this parser to the *pLambda* parser is *Parser* $((g :+: \mathit{Expr}) :+: \mathit{Expr})$, which breaks the automatic injection. Automatic injection is broken, because the *Expr* data type is present at two locations in the composition. Thus it is unclear for the injection if the *Inl* or *Inr* constructor must be used, because both are valid.

To solve this, an alternative definition for *pLambda* can be given that must be used when the *Expr* type is already subsumed by f:

$$\mathit{plambda}' :: \mathit{Functor}\ f, \mathit{Expr} :<: f, \mathit{IsExpr}\ f \Rightarrow \mathit{Parser}\ f \rightarrow \mathit{Parser}\ f$$

which is our original type constrained definition. However, when multiple parsers arguments are accepted, one extra definition is not enough. To illustrate, let us extend our *Lambda* parser by taking two parser as arguments, one for lambda bodies and the other for function application.

$$\begin{aligned}&\mathit{plambda}' :: (\mathit{Functor}\ f, \mathit{Expr} :<: f, \mathit{IsExpr}\ f, \mathit{Functor}\ g, \mathit{IsExpr}\ g, \mathit{Expr} :<: (f :+: g :+: \mathit{Expr}))\\ &\quad \Rightarrow \mathit{Parser}\ f \rightarrow \mathit{Parser}\ g \rightarrow \mathit{Parser}\ (f :+: g :+: \mathit{Expr})\end{aligned}$$

Now, the resulting parser contains terms of type f, terms of type g, and *Expr* terms. Having three types that are composed results in more chances of overlapping compositions, since f can overlap with g, f can overlap with *Expr*, and g can overlap with *Expr*. Furthermore, f and g can overlap on different levels: they can be equal, the can overlap on one term, or they can overlap on multiple terms. For example, we can have $f = i :+: j :+: k$ and $g = h :+: l :+: k$, where f does not subsume g, or the other way, but both contain k, making their composition ambiguous. Of course, the depth where such a duplicate occurs is unknown, thus writing generic combinators is difficult because every possible level must be defined.

To solve this, I introduce a flattening type family that takes two signature compositions, composes the signatures, and flattens the resulting composition, such that duplicates are removed.

```
type family F (f :: * → *) (g :: * → *) where
  F f f = f
  F (f1 :+: f2) g = F f1 (F f2 g)
  F f (g1 :+: g2) = EmbedIfNotFound (Elem f (g1 :+: g2)) f (g1 :+: g2)
  F f g = f :+: g
type family EmbedIfNotFound (e :: Emb) (f :: * → *) (g :: * → *) where
  EmbedIfNotFound (Found _) _ g = g
  EmbedIfNotFound _ f g = f :+: g
```

The type family defines four possible conditions in which the two signatures can be in. The first condition states that when the two signatures are identical, the flattening is just the signature, and not the composition. The second condition is hit when the first signature is a composition of signatures. In this case, the compound signature is extracted into its left and right parts, then the type family is applied on the right part and the second signature, and the type family is applied on the result of that application and the left part of the extraction. The third condition is hit when the second signature is a composition of other signatures. In that case, the type family searches for the first signature in the second signature using *Elem* type family exported by the compdata library, and if not found composes the first signature with the second. The last case is when the two signatures are not a composition of other signatures and are not identical, which results in a composition of the two signatures.

Now, it is possible to redefine the constrained *Lambda* parser without requiring multiple definitions.

```
pLambda :: Functor f, Expr :<: (F f Expr), IsExpr f ⇒ Parser f → Parser (F f Expr)
```

However, because F is a type family, it is not injective and can not be on the left-hand side in a type family definition. As a result, some recursive parser definitions become impossible for the type checker to check. To illustrate, let us again take the example of the lambda parser with two parser arguments, where a new variant (*pLambdaV*) is defined that takes one argument and uses *plambda′* to encode a parser definition.

```
pLambda′ :: Parser f → Parser g → Parser (F3 f g Expr)
plambdaV :: Parser f → Parser (F f Expr)
plambdaV pmain = plambda′ pmain (plambdaV pmain)
type F3 f g e = F (F f g) e
```

In this case, the second parser to *plambda′* is the *plambdaV* parser. The resulting type of *plambdaV* is *Parser* (*F f Expr*), but this type is obtained via the result type of the *pLambda′* parser, which is of type *Parser* (*F3 f g Expr*) where the second argument, *g*, is in this case determined by the type of *pLambdaV*. Hence, the type of *pLambda′* is *Parser* (*F3 f* (*F f Expr*) *Expr*), which is incompatible with the type of *pLambdaV*, which is *Parser* (*F f Expr*), and Haskell signals a type error. Furthermore, because the *pLambdaV* parser is an argument to the *pLambda′* parser, the type of *plambdaV* can not be changed, since a change in the type of *pLambdaV* also changes the type of *pLambda′*.

To solve this, a type level equality constraint — defined with the ∼ operator — must be specified to signal to Haskell that the resulting type of *pLambda′* is equal to the resulting type of *pLambdaV*.

```
plambdaV :: (F f Expr∼F3 f (F f Expr)) ⇒ Parser f → Parser (F f Expr)
plambdaV pmain = plambda′ pmain (plambda″ pmain)
```

Thus, it is now possible to define recursive parsers with type level constraints to guide the composition of higher order parsers to prevent the creation of incompatible parser combinations. This method enables constraint language composition, making the approach to construct exploratory polyglot systems via language composition safer.

8

Discussion

8.1 Exploring interpreters

With the unified model, presented in chapter 5, a consistent exploratory experience is provided and by combining the different behaviours in one model, the different exploratory styles discussed in chapter 4 are possible in the same exploratory session. Furthermore, the language requirements are minimal and enable a wide range of languages to support exploratory programming with little effort, as seen in the case-studies.

The implementation makes no assumptions about the interface used to perform exploratory programming, as showcased via the different eFLINT interfaces. This enables the creation of highly language specific interfaces that provide effective exploratory programming for that language. Furthermore, being decoupled from the interface, construction of more non-developer interfaces, like mage [41], with some support for exploratory programming is possible.

The operations provide the basic building blocks to operate on the exploration tree, allowing an interface to construct more complex interactions. For example, an interface can provide the functionality to execute a program on a group of configurations, which can be fully encoded as combination of **execute** and **jump** operations. Combinations of **execute** and **jump** can also be used to overlay traces or make a small modification in an existing trace and re-execute the trace from that point with the modification. Similar usage of combinations of exploring interpreter operations was showcased with buttons to operate on a configuration in the eFLINT web-interface.

In addition, reproducibility is supported by the implementations up to the point of the environment. reproducibility is an important concept in notebooks, and exporting reproducible exploration sessions makes it possible to share exploration sessions between users and possibly enable pair exploratory programming, where multiple users work together in the same exploration session. To fully enable reproducibility, the reproducibility provided by the exploring interpreter can be combined with tools to construct reproducible environments [42], like Docker [43] and Nix [44].

Finally, the implementation uses an interpreter that evaluates the supplied program in one go. Such an interpreter is also known as a big-step interpreter. Alternatively, there is the small-step interpreter, which evaluates is a program in a combination of small steps [45]. An exploring interpreter implementation defined over a small-step interpreter can make every step a new transition in the exploration tree. This gives a finer grained exploration experience with more insight in the evaluation of a program, opening up integration with debugging. However, a small-step interpreter increases the number of configurations significantly and requires more operations on the exploring interpreter when evaluating, which can impact the performance compared to a big-step interpreter. More research is required to determine how fruitful exploration with a small-step interpreter is, how big the performance penalty is, and what the effects on the flexibility of the exploring interpreter are.

8.2 Language parametric interfaces

With the language parametric interfaces for exploratory programming, languages receive exploratory programming interfaces almost for free, as showcased with the polyglot REPLs, further reducing the required effort for a language to support exploratory programming. Furthermore, via the exploring interpreter protocol, duplicate implementations are reduced and re-usability of interfaces is promoted.

This thesis was focused on the implementation level and not the interface level. Hence, more research is required in the construction of complex interfaces on top of the exploring interpreter protocol. Nonetheless, the exploring interpreter protocol makes it possible to research a generic non-embedded interface that can be re-used by different exploring interpreter implementations and is extensible by users. Such an interface would provide a base to support languages generically and enable users to extend the interface with language specific features. Enabling the user to extend the interface puts the exploratory display in full control of the user, alleviating the language implementer to implement interfaces. Putting the user in full control allows a user to provide display implementations for user specific exploration behaviours, without having to implement all user specific exploration behaviours in one interface. Furthermore, a library can be developed for the creation of components, where some components can be language parametric, allowing users to compose components to create their preferred interface.

8.3 Polyglot language composition

With the method outlined in chapter 7 it is possible to create polyglot systems that support exploratory programming, like REPLs, via language composition. The approach provides a very flexible form of language composition, enabling both fine- and coarse-grained compositions and REPLs. By choosing funcons to translate semantics, effort required to support using language in the system is reduced, because of the high abstraction level funcons provide, and there is a lot of flexibility in freely mixing languages constructs, as shown in the case of *Lambda* and *While*. However, the flexibility also makes it possible that compositions are not well defined. With the type level constraints, it is more difficult to create compositions that are not well defined, but it is still required that the type level constraints are implemented and not overridden. In addition, using type level programming significantly increases the compilation time.

The parsers are based on GLL parsing, which enables parsing of all context free parsers and ambiguities in the grammar are handled by returning multiple parse results. Currently, when a parse has multiple results, the first result is taken. Taking the first result hides the ambiguity from the user. Hiding the ambiguity from the user is nice, because it requires no interaction from the user to continue the exploration session in a polyglot system. However, when the parser results are not deterministic, it can really confuse a user, and when used in agile language development, the ambiguity is necessary information for the language developer. In the last case, a system might be developed that provides the user with a choice to choose the right parser result, and then adapt the existing parsers to take into account the choice made. This way, by making choices in an exploratory polyglot session, the language composition becomes less ambiguous.

With the provided method, several options for future work as possible. A component library of small languages and a combinator library for term parser can be constructed. With these libraries, it becomes possible to compose languages using really small components, and possibly performing these compositions in an exploratory environment, enabling exploratory language development. Furthermore, an extensive evaluation of the method can be performed by building some popular languages with the method and creating a polyglot environment for these languages. For example, a polyglot environment mixing C and Javascript, adding first class support to C for working with JSON.

9

Conclusion

Exploratory programming is a useful programming method when the goal worked towards is not fully known. Usage of exploratory programming is seen in a wide variety of domains. However, current tooling only enables shallow forms of exploratory programming or are focused on the text level, preventing interactivity.

In this thesis, I presented an implementation of a generic exploring interpreter, which enables adding interactive exploratory programming to a wide variety of languages without much effort from the language implementation. Via an evaluation of the exploration experience with the model, a new unified model was proposed that supports a wider range of languages, provides a consistent exploratory experience, and supports a broad range of different exploratory programming styles.

In addition, language parametric interfaces for exploratory programming were created, abstracting away the common functionality and simplifying the creation of exploratory programming interfaces, reducing the required effort to add exploratory programming to languages even further. For non-embedded interfaces, a protocol for exploratory programming was designed and implemented, enabling an interface to work with any exploring interpreter server that implements the protocol, promoting re-usability of interfaces.

Finally, by combining funcons and data types á la carte on top of the language parametric interfaces and the unified model, a method for the creation of flexible polyglot exploratory environments via language composition was created. Via higher order GLL parsers, the method supports the creation of both fine- and coarse-grained exploratory polyglot REPLs, and via type level constraints, the creation of invalid compositions is reduced. With the method, a step is made in to the direction of exploratory language development.

Appendix A

Exploring Interpreter Protocol

This section contains the full definition of the language independent protocol for exploratory programming. The request message, response message, and response error objects are defined by the Language Server Protocol[1].

```
/** JSON RPC error codes */
type ParseError = -32700;
type InvalidRequest = -32600;
type MethodNotFound = -32601;
type InvalidParams = -32602;
type InternalError = -32603;

/** Exploratory programming error codes */
type ReferenceNotInTree = 1;
type ReferenceRevertInvalid = 2;
type ProgramParseError = 3;
type PathNonExisting = 4;
type MetaCommandError = 5;

type DefaultErrorCodes = ParseError | InvalidRequest | MethodNotFound
    | InvalidParams | InternalError;

interface DefaultError extends ResponseError {
    code: DefaultErrorCodes;
}

interface Edge {
    source: uinteger;
    target: uinteger;
    label: EdgeLabel;
}
```

[1]https://microsoft.github.io/language-server-protocol/specifications/specification-current

```
interface EdgeLabel {
    program: string;
    output: string;
}

interface ExecutionTree {
    current: uinteger;
    references: [uinteger];
    transitions: [Edge];
}

interface JumpRequest extends RequestMessage {
    method: "jump";
    params: JumpParams;
}

interface JumpParams {
    reference: uinteger;
}

interface JumpResponse extends ResponseMessage {
    result?: null;
    error?: JumpError;
}

interface JumpError extends ResponseError {
    code: DefaultErrorCodes | ReferenceNotInTree;
}

interface RevertRequest extends RequestMessage {
    method: "revert";
    params: RevertParams;
}

interface RevertParams {
    reference: uinteger;
}

interface RevertResponse extends ResponseMessage {
    result?: [uinteger];
    error?: RevertError;
}

interface RevertError extends ResponseError {
    code: DefaultErrorCodes | ReferenceNotInTree | ReferenceRevertInvalid;
}
```

```
interface ExecuteRequest extends RequestMessage {
    method: "execute";
    params: ExecuteParams;
}

interface ExecuteParams {
    program: string;
}

interface ExecuteResponse extends ResponseMessage {
    result?: ExecuteResult;
    error?: ExecuteError;
}

interface ExecuteResult {
    reference: uinteger;
    output: string;
}

interface ExecuteError extends ResponseError {
    code: DefaultErrorCodes | ProgramParseError;
}

interface DerefRequest extends RequestMessage {
    method: "deref";
    params: DerefParams;
}

interface DerefParams {
    reference: uinteger;
}

interface DerefResponse extends ResponseMessage {
    result?: object;
    error?: DerefError;
}

interface DerefError extends ResponseError {
    code: DefaultErrorCodes | ReferenceNotInTree;
}

interface ExecutionTreeRequest extends RequestMessage {
    method: "getExecutionTree";
}

interface ExecutionTreeResponse extends ResponseMessage {
    result?: ExecutionTree;
    error?: DefaultError;
}

interface TraceRequest extends RequestMessage {
    method: "getTrace";
}

interface TraceResponse extends ResponseMessage {
    result?: [Edge];
    error?: DefaultError;
}

interface PathRequest extends RequestMessage {
    method: "getPath";
    params: PathParams;
}

interface PathParams {
    source: uinteger;
    target: uinteger;
}

interface PathResponse extends ResponseMessage {
    result?: [Edge];
    error?: PathError;
}

interface PathError extends ResponseError {
    code: DefaultErrorCodes | PathNonExisting;
}

interface CurrentReferenceRequest extends RequestMessage {
    method: "getCurrentReference";
}

interface CurrentReferenceResponse extends ResponseMessage {
    result?: uinteger;
    error?: DefaultError;
}

interface AllReferencesRequest extends RequestMessage {
    method: "getAllReferences";
}

interface AllReferencesResponse extends ResponseMessage {
    result?: [uinteger];
    error?: DefaultError;
}

interface LeavesRequest extends RequestMessage {
    method: "getLeaves";
}

interface LeavesResponse extends ResponseMessage {
    result?: [uinteger];
    error?: LeavesError;
}

interface LeavesError extends ResponseError {
    code: DefaultErrorCodes | ReferenceNotInTree;
}

interface MetaRequest extends RequestMessage {
    method: "metaCommand";
}

interface MetaResponse extends ResponseMessage {
    result?: object;
    error?: MetaError;
}

interface MetaError extends ResponseError {
    code: DefaultErrorCodes | MetaCommandError;
}
```

References

[1] Beau Sheil. **Datamation®: Power Tools for Programmers**. In *Readings in artificial intelligence and software engineering*, pages 573–580. Elsevier, 1986. 1

[2] Mary Beth Kery and Brad A. Myers. **Exploring exploratory programming**. In Austin Z. Henley, Peter Rogers, and Anita Sarma, editors, *2017 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2017, Raleigh, NC, USA, October 11-14, 2017*, pages 25–29. IEEE Computer Society, 2017. 1, 5

[3] Caitlin Kelleher, Randy F. Pausch, and Sara B. Kiesler. **Storytelling alice motivates middle school girls to learn computer programming**. In Mary Beth Rosson and David J. Gilmore, editors, *Proceedings of the 2007 Conference on Human Factors in Computing Systems, CHI 2007, San Jose, California, USA, April 28 - May 3, 2007*, pages 1455–1464. ACM, 2007. 1

[4] Ben Fry. *Visualizing data - exploring and explaining data with the processing environment*. O'Reilly, 2008. 1

[5] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E. Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B. Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, Carol Willing, and Jupyter Development Team. **Jupyter Notebooks - a publishing format for reproducible computational workflows**. In Fernando Loizides and Birgit Schmidt, editors, *Positioning and Power in Academic Publishing: Players, Agents and Agendas, 20th International Conference on Electronic Publishing, Göttingen, Germany, June 7-9, 2016*, pages 87–90. IOS Press, 2016. 1

[6] Hans Fangohr, Thomas Kluyver, and Massimo DiPierro. **Jupyter in Computational Science**. *Comput. Sci. Eng.*, **23**(2):5–6, 2021. 1

[7] YoungSeok Yoon and Brad A. Myers. **An exploratory study of backtracking strategies used by developers**. In Helen Sharp, Yvonne Dittrich, Cleidson R. B. de Souza, Marcelo Cataldo, and Rashina Hoda, editors, *5th International Workshop on Co-operative and Human Aspects of Software Engineering, CHASE 2012, Zurich, Switzerland, June 2, 2012*, pages 138–144. IEEE Computer Society, 2012. 1, 26

[8] Robert DeLine and Danyel Fisher. **Supporting exploratory data analysis with live programming**. In Zhen Li, Claudia Ermel, and Scott D. Fleming, editors, *2015 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2015, Atlanta, GA, USA, October 18-22, 2015*, pages 111–119. IEEE Computer Society, 2015. 1

[9] L. Thomas van Binsbergen, Mauricio Verano Merino, Pierre Jeanjean, Tijs van der Storm, Benoit Combemale, and Olivier Barais. *A Principled Approach to REPL Interpreters*, pages 84–100. ACM, 2020. 1, 6, 15, 17, 58

[10] Kayur Patel, James Fogarty, James A. Landay, and Beverly L. Harrison. **Investigating statistical machine learning as a tool for software development**. In Mary Czerwinski, Arnold M. Lund, and Desney S. Tan, editors, *Proceedings of the 2008 Conference on Human Factors in Computing Systems, CHI 2008, 2008, Florence, Italy, April 5-10, 2008*, pages 667–676. ACM, 2008. 1

[11] Mary Beth Kery, Amber Horvath, and Brad A Myers. **Variolite: Supporting Exploratory Programming by Data Scientists.** In *CHI*, **10**, pages 3025453–3025626, 2017. 2, 13

[12] Hiroaki Mikami, Daisuke Sakamoto, and Takeo Igarashi. **Micro-Versioning Tool to Support Experimentation in Exploratory Programming**. In Gloria Mark, Susan R. Fussell, Cliff Lampe, m. c. schraefel, Juan Pablo Hourcade, Caroline Appert, and Daniel Wigdor, editors, *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems, Denver, CO, USA, May 06-11, 2017*, pages 6208–6219. ACM, 2017. 2, 13

[13] Pavneet Singh Kochhar, Dinusha Wijedasa, and David Lo. **A Large Scale Study of Multiple Programming Languages and Code Quality**. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering,*

SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1, pages 563–573. IEEE Computer Society, 2016. 2

[14] David Chisnall. **The challenge of cross-language interoperability**. *Commun. ACM*, **56**(12):50–56, 2013. 2

[15] David W. Sandberg. **Smalltalk and exploratory programming**. *ACM SIGPLAN Notices*, **23**(10):85–92, 1988. 5

[16] Erik Sandewall. **Programming in an Interactive Environment: the LISP Experience**. *ACM Comput. Surv.*, **10**(1):35–71, 1978. 5

[17] Urs Holzle. *Adaptive optimization for SELF: reconciling high performance with exploratory programming*. PhD thesis, Stanford University, 1994. 5

[18] Wouter Swierstra. **Data types à la carte**. *J. Funct. Program.*, **18**(4):423–436, 2008. 7

[19] Philip Wadler et al. **The expression problem**. *Posted on the Java Genericity mailing list*, 1998. 7

[20] Erik Meijer, Maarten M. Fokkinga, and Ross Paterson. **Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire**. In John Hughes, editor, *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings*, **523** of *Lecture Notes in Computer Science*, pages 124–144. Springer, 1991. 8

[21] Patrick Bahr and Tom Hvitved. **Compositional data types**. In Jaakko Järvi and Shin-Cheng Mu, editors, *Proceedings of the seventh ACM SIGPLAN workshop on Generic programming, WGP@ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 83–94. ACM, 2011. 10

[22] N Watts. **Even more than polyglot programming**. Technical report, Retrieved 2021-07-27, from https://thewonggei.wordpress.com/2008/01/22/even-more-than-polyglot-programming, 2008. 10

[23] Thomas Würthinger, Christian Wimmer, Andreas Wöss, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. **One VM to rule them all**. In Antony L. Hosking, Patrick Th. Eugster, and Robert Hirschfeld, editors, *ACM Symposium on*

New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013, pages 187–204. ACM, 2013. 14

[24] Matija Sipek, Branko Mihaljevic, and A. Radovan. **Exploring Aspects of Polyglot High-Performance Virtual Machine GraalVM**. In Marko Koricic, Zeljko Butkovic, Karolj Skala, Zeljka Car, Marina Cicin-Sain, Snjezana Babic, Vlado Sruk, Dejan Skvorc, Slobodan Ribaric, Stjepan Gros, Boris Vrdoljak, Mladen Mauher, Edvard Tijan, Predrag Pale, Darko Huljenic, Tihana Galinac Grbac, and Matej Janjic, editors, *42nd International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2019, Opatija, Croatia, May 20-24, 2019*, pages 1671–1676. IEEE, 2019. 14

[25] Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. **High-performance cross-language interoperability in a multi-language runtime**. In Manuel Serrano, editor, *Proceedings of the 11th Symposium on Dynamic Languages, DLS 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 78–90. ACM, 2015. 14, 55

[26] Tomas Petricek, James Geddes, and Charles Sutton. **Wrattler: Reproducible, live and polyglot notebooks**. In Melanie Herschel, editor, *10th USENIX Workshop on the Theory and Practice of Provenance, TaPP 2018, London, UK, July 11-12, 2018*. USENIX Association, 2018. 14, 55

[27] Damian Frölich and L. Thomas van Binsbergen. **A Generic Back-End for Exploratory Programming**. In *The 22nd International Symposium on Trends in Functional Programming (TFP 2021)*, **12834** of *LNCS*. Springer, 2021. 15

[28] L. Thomas van Binsbergen. *Executable Formal Specification of Programming Languages with Reusable Components*. PhD thesis, Royal Holloway, University of London, 2019. 15

[29] L. Thomas van Binsbergen, Lu-Chi Liu, Robert van Doesburg, and Tom van Engers. **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. 22

[30] Martin Churchill, Peter D. Mosses, Neil Sculthorpe, and Paolo Torrini. **Reusable Components of Semantic Specifications**. In *Transactions on Aspect-Oriented Software Development XII*, TAOSD 2015, pages 132–179, 2015. 23

[31] Peter D. Mosses. **Software meta-language engineering and CBS**. *Journal of Computer Languages*, **50**:39–48, 2019. 23

[32] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E. John, and Brad A. Myers. **The Story in the Notebook: Exploratory Data Science using a Literate Programming Tool**. In Regan L. Mandryk, Mark Hancock, Mark Perry, and Anna L. Cox, editors, *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems, CHI 2018, Montreal, QC, Canada, April 21-26, 2018*, page 174. ACM, 2018. 25

[33] Oleg Kiselyov and Hiromi Ishii. **Freer monads, more extensible effects**. In Ben Lippmeier, editor, *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, pages 94–105. ACM, 2015. 29

[34] Peter D. Mosses. **Modular Structural Operational Semantics**. *Journal of Logic and Algebraic Programming*, **60–61**:195–228, 2004. 29

[35] Peter D. Mosses and Mark J. New. **Implicit Propagation in Structural Operational Semantics**. *Electronic Notes in Theoretical Computer Science*, **229**(4), 2009. 29

[36] EDWIN BRADY. **Idris, a general-purpose dependently typed programming language: Design and implementation**. *Journal of Functional Programming*, **23**:552–593, 9 2013. 37

[37] Chris Lattner. **LLVM and Clang: Next generation compiler technology**. In *The BSD conference*, **5**, 2008. 46

[38] Richard M Stallman. **GNU compiler collection internals**. *Free Software Foundation*, 2002. 46

[39] Joey Lai. *Supporting Exploratory Programming in Computational Notebooks with an Exploring Interpreter (Working title)*. Master's thesis, Universiteit van Amsterdam, the Netherlands, 2021. 50

[40] L. Thomas van Binsbergen, Elizabeth Scott, and Adrian Johnstone. **GLL parsing with flexible combinators**. In David J. Pearce, Tanja Mayerhofer, and Friedrich Steimann, editors, *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2018, Boston, MA, USA, November 05-06, 2018*, pages 16–28. ACM, 2018. 57

[41] Mary Beth Kery, Donghao Ren, Fred Hohman, Dominik Moritz, Kanit Wongsuphasawat, and Kayur Patel. **mage: Fluid Moves Between Code and Graphical Work in Computational Notebooks**. In Shamsi T. Iqbal, Karon E. MacLean, Fanny Chevalier, and Stefanie Mueller, editors, *UIST '20: The 33rd Annual ACM Symposium on User Interface Software and Technology, Virtual Event, USA, October 20-23, 2020*, pages 140–151. ACM, 2020. 65

[42] Jürgen Cito and Harald C. Gall. **Using docker containers to improve reproducibility in software engineering research**. In Laura K. Dillon, Willem Visser, and Laurie A. Williams, editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*, pages 906–907. ACM, 2016. 66

[43] Dirk Merkel et al. **Docker: lightweight linux containers for consistent development and deployment**. *Linux journal*, **2014**(239):2, 2014. 66

[44] Eelco Dolstra, Merijn de Jonge, and Eelco Visser. **Nix: A Safe and Policy-Free System for Software Deployment**. In Lee Damon, editor, *Proceedings of the 18th Conference on Systems Administration (LISA 2004), Atlanta, USA, November 14-19, 2004*, pages 79–92. USENIX, 2004. 66

[45] Gordon D. Plotkin. **A structural approach to operational semantics**. *J. Log. Algebraic Methods Program.*, **60-61**:17–139, 2004. 66