TESTING STATIC PROGRAM ANALYSES WITH A STATE-COLLECTING MONADIC DEFINITIONAL INTERPRETER

HOANG NGOC TRAM

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2022

Supervisor:

Associate Professor Ilya Sergey

Examiners:

Professor Olivier Danvy

Professor Joxan Jaffar

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

November 25, 2022	Hoang Ngoc Tram
.....	.....
Date	*Name*

Acknowledgements

This work was made possible with the help and support of the following people, to whom I am deeply grateful.

Professor Ilya Sergey's patience and commitment to teaching allowed me and many other students to realise and then actualise our passion for the field. I am thankful for the opportunities he generously provides and for his tireless guidance, advice, and wisdom throughout the past four years. I am also grateful to the faculty of the School of Computing and of Yale-NUS College. Through their patient, kind, and effective teaching, Professor David Smith and Professor Francesca Spagnuolo share their love for mathematics; they instill into their students the importance of sustained work, rigour, and paying close attention to detail. In the Functional Programming and Proving lectures, Professor Olivier Danvy challenges his students with difficult but interesting questions that bring out their curiosity; he pushes them to do better by making them believe in themselves. The faculty and staff of these institutions never fail to make students feel like they are a part of a learning community, and for that I am grateful too.

To Yunjeong, Kiran, and Leyli, thanks for your scientific interest and for your support and comments. You have been incredibly kind and patient, and this dissertation truly would not have been the same without you.

To Alysha, Denise, Raya, and Emma, thanks for being pillars of sanity throughout our university years and for always filling the room with

laughter, love, and care. Thanks to Nastya, Fiona, and Sangam for insightful conversations, heartfelt support, and for always staying true to yourselves and your passions. Thank you Vivien for your gentleness, for your generous love and care, and for our late night talks about what is important and what is right. And of course, thank you Maxine, for your continuous cheering and, through thick and thin, for always being the light at the end of the tunnel.

To my family, thanks for always believing in me. I love you back endlessly.

Contents

Declaration i

Acknowledgements ii

Contents iv

Summary vi

List of Figures viii

1 Introduction 1

1.1 Approach and Goal 2

1.2 Why Scilla? 3

1.3 Contributions and Outline 3

2 Background 5

2.1 Definitional Interpreter 5

2.2 Static Analysis 8

2.2.1 Data Flow Analysis 10

2.2.2 Control Flow Analysis 12

2.3 Monads 13

2.4 Overview of Scilla 14

2.4.1 Scilla's Monadic Interpreter 16

3 Embedding the Harness for Collecting Semantics 20

3.1 Modularity of Semantics Collection 20

4 Specialised Collection and Case Studies 23

4.1 Testing Type Conformance 23

4.1.1 Collection of Type Flows into Identifiers 24
4.1.2 Results of Testing Type Conformance 29
4.2 Testing the Type-Flow Analysis 30
4.2.1 Collection of Type Flows into Type Variables 32
4.2.2 Results of Testing the Type-Flow Analysis 33
5 Related Work **35**
6 Discussion **37**
6.1 Extensibility and Future Work 37
6.2 Conclusion 38
Bibliography **40**

Summary

Static analyses are commonly applied as a foundation for code optimisations as well as for the detection of safety and security gaps in software systems [Bug+18; Shi91; Mig10]. To reliably serve these purposes, static analyses must soundly predict a program's properties. Although it is possible to formally prove the soundness of an analyser's design, in practice, analysers can be large and complex making it difficult to ensure the absence of implementation bugs. In an attempt to address this issue, this project presents an approach of utilising state-collecting monadic interpreters in order to test static properties derived from static program analyses. While this approach attempts at finding existent bugs, it does not serve as a proof of complete absence of bugs. As Dijkstra famously said: "Program testing can be used to show the presence of bugs, but never to show their absence."

Static analyses produce an over-approximation of a program's run-time behaviour [NNH04; Shi91; AC76]. One can test these over-approximations by recording a program's run-time states and checking if the analyser's results are indeed a correct approximation. In other words, given the static analyses' predicted set of states a program may encounter, one can check if the states the program reaches are indeed in the set by evaluating the program and dynamically collecting intermediate executions.

To this end, we propose the use of a *monadic definitional interpreter* equipped with *collecting semantics* [Rey98b; Dar+17]. While the definitional interpreter evaluates the program, the monadic data type allows the

collection of necessary information at every intermediate execution [LHJ95; Ser+13; BHM00]. We can then check whether the dynamic concrete properties are a subset of the static abstract properties.

In the project, we first present the process of refactoring a production-scale definitional interpreter of a smart-contract language Scilla to obtain dynamic collecting capabilities. Scilla is a functional ML-style language that combines pure functional calculus based on System F and imperative programming [Ser+19; SKH18].

Having successfully collected some dynamic properties, we present two case studies of testing *static type conformance* and *static type-flow analysis*. Testing *static type conformance* means checking whether the static type-checker's inferred types of functional expressions are consistent. Additionally, in the context of imperative programs, testing type-conformance implies checking that the intermediate variables and function parameters are only assigned values of their declared type. Afterwards, we test the *static type-flow analysis* implemented as the basis of the monomorphisation optimisation pass in the Scilla to LLVM compiler [Nag+20]. First, we test the two mentioned properties on test programs in the Scilla repository. We then work with a random Scilla program generator that allows us to perform a more extensive test case study.

List of Figures

2.1 Example of Flow Graph. 10

2.2 Monad Type Signature in OCaml. 13

2.3 Maybe Monad in OCaml. 13

2.4 Implementation of the `FungibleToken` contract in Scilla. 16

2.5 Snippet of Scilla's Expression Interpreter 17

4.1 Trace of Run-Time Type Flows 27

1 Introduction

The demand for static program optimisations is steadily growing as features, such as *polymorphism*, are introduced into existing languages at a real performance penalty [EP17; Shi91; Wee06; Pał+11]. The static correctness guarantee of such optimisations is only as good as their implementations. In other words, a single bug in a compilation pass or even static type-checkers can defy the whole purpose of a strong type-system and violate run-time guarantees [Hoa+22; Ser+19]. For example, for smart contract languages which statically enforce properties, such as preservation of assets [Bla+19] or statically checked communication protocols [Dar+17], the inability to uphold static guarantees can easily be exploited by adversarial entities for monetary profit [Nag+20; SKH18]. [7]r0.35

```
1  read(a);
2  if (a > 42) {
3      x = 1;
4  } else {
5      x = 0; //Not dead code
6  }
```

As such, the consequences of having bugs in a program analysis tool, such as announcing non-existent bugs, failing to locate program bugs, or silently introducing bugs, can be costly. For example, a bug in an optimising static analysis such as *dead-code elimination* can occur in a program seen in Figure 1, if the analysis misclassifies Line 5 as dead code (code that, upon removal, does not affect program's results [KRS94]) when

is is not safe to do so. In the Figure, there is no guarantee that variable a is greater than `42`, and so deeming Line 5 as dead code to then be removed is a bug in the analysis.

1.1 Approach and Goal

Following this demand for static program optimisations, this project proposes a method of testing static analyses that starts with *definitional interpreters* [Rey72; MRL13; AR17; Rey98b]. In essence, a definitional interpreter, implemented for some language `X`, is a function that evaluates a program written in language `X`. Furthermore, definitional interpreters give a simple, clear, and *concrete* account of program semantics [Rey98b; MRL13]. On the other hand, static program analysis *abstracts* runtime behaviour opting to provide an *over-approximated* account of the program semantics in return for computability [Shi91; NNH04]. In other words, given a program has a runtime property `A`, a static-analysis can predict that the program's property is among the set of properties `{A, B, C}`, i.e., an over-approximation of the program's run-time behaviour. As such, one can then test the abstract set of results of static program analysers by checking whether they are a super-set of a definitional interpreter's concrete set of results. Considering the example program in Figure 1: if the static dead-code analysis states that Line 5 is safe to remove, one can evaluate the program using a definitional interpreter which can record that Line 5 was indeed encountered and is, therefore, not safe for removal. From this example, we see how given a static assumption, a concrete intermediate execution state can prove the assumptions false.

In order to obtain concrete intermediate executions, our project equips the interpreter with a static-collecting *monad* [BHM00; Ser+13; Fil94]. Monads, originally adapted to encode computational effects in purely

functional languages, offer the ability to reinterpret computations such as making certain computational effects hidden or fully explicit. Therefore, retrofitting a definitional interpreter with a monad offers modular embedding of new properties such as intermediate state collection [LHJ95; Dar+17]. This project aims to provide a proof-of-concept of testing a constraint-based higher-order program analysis by refactoring a production-scale definitional interpreter for Scilla.

1.2 Why Scilla?

Scilla is a smart-contract language first introduced in 2018 and has since been widely researched and used for smart contract and even game implementations [SKH18; Ser+19]. Scilla combines pure functional properties based on System F, imperative state-managing computations, and message-passing semantics for contract communications. Additionally, its semantics are defined with a monadic definitional interpreter written in OCaml, streamlining the process of modularly implementing a state-collecting harness. Additionally, Scilla designers recently released a Scilla to LLVM compiler featuring constraint-based static program optimisations which we can test with the collecting evaluator [Nag+20]. Finally, Scilla is deployed on top of Zilliqa, a real-world blockchain, highlighting the significance of the implementation of Scilla being bug-free as any bugs in the implementation can have real-world implications.

1.3 Contributions and Outline

In the chapters to follow, we present three key contributions of this project:

- We devise a monad for collecting the semantics of a System F based language.

- We incorporate the collecting harness into Scilla's continuation-passing style monadic interpreter (written in OCaml).
- We provide an evaluation of our harness by testing static type conformance and static type-flow analysis on arbitrary Scilla programs.

The remainder of the report is structured as follows. Chapter 2 provides an overview of the technical background of this project and explains the concepts of definitional interpreters, static analyses, monads, and how Scilla is implemented. Chapter 3 details the process of modularly implementing state collecting semantics into Scilla's definitional interpreter. Chapter 4 evaluates our concept of testing static analyses by testing Scilla's type conformance and static type-flow analysis and it presents our findings. Chapter 5 describes related work regarding other methodologies of testing static analyses and how our experiments fit into the current literature. Finally, Chapter 6 concludes the report by discussing the extensibility of this approach and future research ideas.

2 Background

This chapter provides an overview of components essential to this thesis, namely definitional interpreters (2.1), static analyses (2.2), monads (2.3), and Scilla (2.4).

2.1 Definitional Interpreter

This thesis explores the idea of refactoring a *definitional interpreter* in order to allow dynamic collection of a program's properties. As such, it is important that we first introduce what a definitional interpreter is as well as review some of its past literature.

Many aspects of simple applicative programming languages can be defined using logic, meaning one can construct a logical statement that describes the relations between inputs and outputs of a program [Flo93]. However, while this is true for some properties of first-order languages, the same approach cannot be applied for higher-order languages. Instead, as Reynolds has demonstrated, higher-order languages can be defined with *definitional interpreters* that are written in a second, potentially better understood, language [Rey72].

Before discussing what a definitional interpreter is, it is important to make a distinction between the *defined* and the *defining* language, in which the *defined* language is the language whose semantics are described via

the definitional interpreter written in the *defining* language. Therefore, a definitional interpreter is essentially a function, written in a *defining* language, that evaluates programs in the *defined* language [Rey98b; AR17; Rey72].

Given a functional programming language, its definitional interpreter defines *expressions*, i.e., meaningful programs, how expressions are to be *evaluated*, and, subsequently, what *values* these expressions result in. For example, a definitional interpreter can describe how an expression $1 + 2$ can be evaluated to get the value 3. However, an expression such as $x + y$ can be evaluated to different values depending on what x and y denote at the point of evaluation. Therefore, the definitional interpreter also specifies an *environment* that stores values bound to variables.

The simplest expressions are *constants* and *variables*, where *constants*, such as integer 1 or boolean `true`, are evaluated to be themselves and *variables* are evaluated to the value they are bound to in the environment. An applicative language may also have *functions* and, thus, function *applications* as expressions. For example, $f(a_1, ..., a_n)$ describes a function application of some function f applied to n arguments $a_1, ..., a_n$. Given a language that has functions, it must have expressions that evaluate to functions, namely *lambda abstractions*. An example of a lambda abstraction can be the expression $\lambda x_0, ..., x_n.r$ which consists of n *parameters* $x_0, ..., x_n$ and a *body* r [Rey72; Pie02].

The definitional interpreter outlines every semantic definition of a language starting general definitions mentioned above down to specific details of whether the language evaluates partial function applications, whether it is call-by-value or call-by-name, and other similar properties. It is interesting to note how some properties of the defined language is not user defined but rather influenced by the defining language as definitional interpreters are *leaky* abstractions [Rey72; DN01]. The term *leaky* here

describes how the defined language gains a property of the defining language through the definitional interpreter without the user explicitly implementing the property itself. This is especially true for *meta-circular* interpreters which define each component of the defined language with a corresponding component of the defining language [Rey72]. Although meta-circular interpreters are considered the most concise and simple approach to writing an interpreter, a meta-circular interpreter can lead to misunderstandings of the defining language carrying over into the defined language as well as difficulties in extending the defined language with new features.

To work around these issues, Reynolds shows how different tools can be used to enhance a definitional interpreter. Reynolds explains how one can transform an interpreter to be free of defining language's properties, specifically first-class functions through *defunctionalisation* and evaluation order through the *continuation-passing-style (CPS) transformation* [Rey72]. *Defunctionalisation* describes a process of replacing each function space with a data type which enumerates possible function abstractions that may arise at a specific point of a program coupled with an apply function [DN01]. Through defunctionalisation, one can transform whole higher-order programs into first-order programs, where functions can no longer be passed as arguments or returned as results. On another hand, *CPS transformation* describes the process of introducing continuations to name the intermediate results of a term and explicitly sequentialise the computations [DF92]. Therefore, a CPS transformation provides a "degree of freedom" that allows to meet the condition of independence from the order-of-application of the defining language [Rey72].

Building on Reynold's work, Ager et al. applies the results of Reynolds study to derive semantics where control contexts are built upon the defining language's contexts (evaluators), first-class functions (continuations),

and data (defunctionalised continuations) [Age+03]. Midtgaard et al. then contribute to the body of this scholarship by showing how interpreter performance can improve depending on implementations of low-level representations, addressing the issue of slow performance associated with definitional interpreters [MRL13].

In the context of this thesis, the noteworthy detail about definitional interpreters is that through the evaluation of a program, a definitional interpreter returns a corresponding *concrete* result. The use of the term "concrete" here describes how there is no ambiguity about the execution of the program and its result. Fortunately, definitional interpreters written in monadic style allow for a wide variety of collection of the concrete semantics such as trace semantics, i.e., the collection of streams of states the interpreter reaches, or the collection of dead code [Dar+17]. Having collected the concrete semantics, one can then test their over-approximating abstract semantics (collected by *static analyses*) by checking whether the abstractions include the concrete data seen by the interpreter.

2.2 Static Analysis

Having discussed definitional interpreters, let us provide an overview of what an interpreter will be testing in this thesis, namely *static analyses*. In this chapter, we elaborate on what a static analysis is, what kind of static analyses there are, and what purpose they serve. This chapter aims to provide a general study of static analysis to allow the readers, who are not yet familiar, to familiarise themselves with the topic.

Static analyses are commonly applied as a basis for code optimisations and detection of safety and security gaps of software systems [Bug+18; Shi91; Mig10]. In the context of optimisations, the static analyses' aim is to generate code without redundant or superfluous computations, allowing

the generated program to be executed more efficiently. To allow for these optimisations, they must soundly predict the properties of the programs they analyse. These predictions occur at compile-time meaning the programs are not evaluated which is why these analyses are considered "static" [Shi91; NNH04].

The common theme of static analysis is that having not evaluated the program, the possibilities of what values and behaviours may arise of the program can be infinite. In order to keep the analysis computable, one can only provide over-approximating results. This can be seen in the following example taken from Nielson's et al. *"Principles of Program Analysis"* book:

```
read(x); (if x > 0 then y := 1 else (y := 2; S)); z := y
```

where `S` is some statement that does not reassign to `y` [NNH04].

Looking at the program, one can deduce that the only assignments to `y` that can reach the statement `z := y` are assignments of `1` and `2`. When running the program, it may be the case that only assignment `y := 1` can reach `z := y` because `S` does not terminate given $x \leq 0$ and $y = 2$. However, since the property of `S` terminating is undecidable, the analysis is more likely to give a safer approximation of the program behaviour, namely either `1` or `2` will be assigned to `z`. The analysis may return an even safer approximation, stating that an integer will be assigned to `z`, which we can still accept. An analysis result like that, however, provides less useful information whereas we prefer a more precise answer, such as a result that states the values assigned are among `1` or `2` (rather than among all possible integers). Clearly, the challenge is not to produce the safest approximation, but rather the most precise, yet still safe, approximation [NNH04].

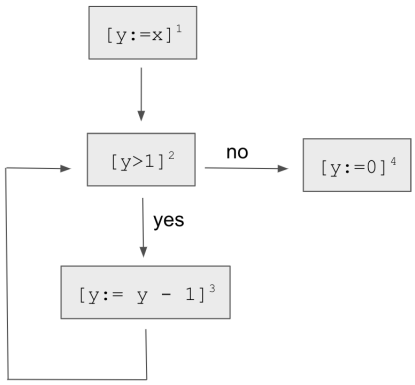

Figure 2.1: Example of Flow Graph.

2.2.1 Data Flow Analysis

The two most well-known static analyses are *Data Flow Analysis* and *Control Flow Analysis*. Although we will not go into implementation details of these analyses, this study aims to provide a high-level understanding as to what they are.

Data Flow Analysis, as per its name, states what useful data reaches certain points of the program. Some classical Data Flow Analyses include Available Expressions (to determine which expressions have been computed at certain points of the program), Reaching Definitions (to determine which variable definitions reach certain points of programs), and Live Variables (to determine which variables are still in use at certain points of programs) [NNH04; AC76].

In Data Flow Analysis, it is customary to think of programs as flow graphs [NNH04; AC76]. Let us consider the following program for an analysis such as Reaching Definitions:

```
y := x; while y > 1 do (y:= y - 1); y := 0
```

Figure 2.1 shows the program translated into a graph where each node is an elementary *block* and each edge describes how *control* might pass between the blocks. Knowing the terms, let us re-define the Reaching Definitions analysis as an analysis that attempts to determine which

variable definitions reach what elementary block in the program flow graph.

There are two ways to approach any Data Flow Analysis, namely *equational approach* and *constraint-based approach* [NNH04; Shi91; AC76]. In the *equational approach*, one can specify different classes of equations by finding the relations between the definitions that exit a block and the definitions that enter the block. For example, the definitions exiting block 1 (1) include the definitions that entered the block and the definition of y in the block, and (2) exclude all of the previous assignments to y as the block contains a new definition of y. This can be seen in the following equation:

$$\mathtt{RD}_{exit}\mathtt{(1)= (RD}_{entry}\mathtt{(1)\backslash\{(y, l)|\ l} \in LAB\mathtt{\})\cup\{(y, 1)\}}$$

where LAB is the set of labels $\{1, 2, 3, 4\}$ of the blocks in the flow graph, $\mathtt{RD}_{exit}\mathtt{(1)}$ and $\mathtt{RD}_{entry}\mathtt{(1)}$ represent definitions entering and exiting block 1 respectively, and the pair `(y, 1)` implies assignment to y at block 1. Using this method, one can write 4 equations that state what definitions leave each block in the graph at Figure 2.1. While this is one way of obtaining equations, it is certainly not the only way.

On the other hand, in the *constraint-based approach*, the idea is to extract constraints or inclusions from the program [NNH04; KSS17; AC76]. Considering the same block as above, the constraint for an assignment block becomes the exclusion of all pairs of `(y, l)` where $\mathtt{l} \in LAB$ from the set of definitions entering block 1, as well as a constraint that states the inclusion of the pair `(y, 1)`. These constraints can be written as

$$\mathtt{RD}_{exit}\mathtt{(1)} \supseteq \mathtt{RD}_{entry}\mathtt{(1)\backslash\{(y, l)|\ l} \in LAB\mathtt{\}}$$

$$\mathtt{RD}_{exit}\mathtt{(1)} \supseteq \mathtt{\{(y, 1)\}}$$

We can rewrite these two constraints to make the connection between the equational and constraint-based methods clearer.

$$\texttt{RD}_{exit}\texttt{(1)} \supseteq \texttt{RD}_{entry}\texttt{(1)}\texttt{\textbackslash\{(y, l)| l} \in LAB\texttt{\}} \cup \texttt{\{(y, 1)\}}$$

It becomes apparent that we obtained a version of the previous equation except the equality sign is replaced with the inclusion sign. Therefore, the solution for the equation based approach is also the solution for the constraint-based approach, whereas the opposite is not always true.

As mentioned above, the most useful static analysis result is the one that is most precise while still being safe. As such, one aims for the solution to the Reaching Definition analysis to be the least solution, i.e., one that contains the fewest pairs of reaching definitions while still being consistent with the program. In this example, we find that both the constraint-based and equational method would conclude with the same least solution [NNH04; KSS17; AC76].

2.2.2 Control Flow Analysis

In the previous section, we introduced what Data Flow Analysis is and provided an example of one to introduce the concepts of equational and constraint-based methods, how constraints are collected, and how the least solution is computed. Constraint based methods, where the user aims to find the least solution, are used for Control Flow Analysis as well. As such, we will not be going through an example of a Control Flow Analysis, but will rather explain the high level idea of the analysis.

The aim of the Control Flow Analysis is to find which elementary block will lead to what other elementary block [NNH04; Shi91]. Let us consider the following example:

```
let f = fun x => x 0 in
let g = fun y => y + 2 in
in f g
```

The program describes a function `f` that expects another function to be bound to its parameter `x`. Evaluating the program shows how the defined

function `g` is bound to the parameter `x` to return the result 2 of the program. Applying `f` transfers control to the body of the function, i.e., `x 0`. The application of `x` transfers control to the body of the function bound to `x`. The problem that the analysis solves is knowing the body of `x`, i.e., what parameters `f` is called with. In other words, the analysis aims to answer the question "For each function declaration, to which functions application do they flow?" [NNH04; Shi91].

In this section, we explained static analysis, its varying types, and how it operates. In the context of this thesis, it is important to understand that static analyses aim to find the most precise approximation of the concrete semantics of a program. One way to test these approximations is to collect these concrete semantics by evaluating the program and checking whether the approximation of the behaviour is a super-set of the collected concrete semantics. A tool that allows us to collect concrete results are *monads*, which we introduce in the following section.

2.3 Monads

```
module type Monad = sig
    type 'a monad
    val return : 'a -> 'a monad
    val (>>=)  : 'a monad -> ('a -> 'b monad) -> 'b monad
end
```

Figure 2.2: Monad Type Signature in OCaml.

```
1  module MaybeMonad = struct
2    type 'a monad  = 'a option
3    let return (x: 'a) : 'a monad = Some x
4    let (>>=) (m: 'a monad) (f: 'a -> 'b monad) : 'b monad =
5      match m with
6      | None -> None
7      | Some x -> f x
8  end
```

Figure 2.3: Maybe Monad in OCaml.

Monads are a function composition technique that originally became popular through their use in Haskell, a purely functional programming

language. They were first adapted to encode computational effects as they allow for explicit data and control flow handling by wrapping effectful intermediary executions [BHM00]. As a result, they can be used to define programming languages' semantics [Ser+13].

Traditionally, a type signature of a monad specifies the use of two polymorphic functions called return and bind (»=), as shown in Figure 2.2 [BHM00; MM17]. For example, the Maybe monad, which can be seen in Figure 2.3, can act as an alternative to handling exceptions - instead of raising errors, our monadic function propagates the result `None`. Otherwise, we would get a result in the form `Some` result. The return function then returns successful output, whereas the bind function propagates the result or None if the function meets an error. This example shows us how monads provide a degree of control to the developer as it allowed for control over the flow of the function by disallowing exceptions, and of control over what data can be returned.

As mentioned in Section 2.1, a definitional interpreter is a structurally recursive function that defines a language's semantics. Through the monad's ability to lift and bind program executions, we gain access to binding desired information to intermediary results, such as intermediary program properties. In other words, with monads, we gain the ability to collect program states modularly, i.e., without interrupting the logic of program execution. One of the interpreters that utilise this monadic property is, in fact, the Scilla interpreter. In the next section, we introduce Scilla and discuss the implementation of its definitional interpreter.

2.4 Overview of Scilla

Scilla is a functional language which combines pure functional calculus based on System F and imperative computations, designed with the

aim to implement smart contracts in the form of state-transition systems that communicate via message passing [Ser+19; SKH18; Hoa+22]. Each contract implementation follows a template, an example of which can be seen in Figure 2.4 showing a definition of the most popular contract used to define fungible tokens. The contract implementation starts with declaring the Scilla version at Line 1, followed by defining a library `FungibleToken` of three pure functions that can be used later in the contract. The user also has the freedom to import pure functions predefined in the Scilla repository or in other contracts. From Line 8 to Line 40 follows the implementation of the `FungibleToken` contract which expects five immutable contract parameters, types of which are explicitly declared. A contract constraint on immutable parameters may be defined at this point to establish a contract invariant for contract deployment. On Lines 11 to 14, the contract initiates two mutable fields, followed by the definition of six transitions from Lines 17 to 40. Transitions can be invoked when an external entity sends a message to the contract which can result in a change in the contract state, an emission of observable events, and an addition of a message to outbox of the contract. As seen in Figure 2.4, the transition `BalanceOf` responds by sending a message upon the end of the transition containing the balance of a specific token owner. The user may also define procedures which, in contrast to transitions, can only be invoked when called from a transition and not from externally received messages.

Transitions or procedures may apply defined or imported pure functions as seen in transition `BalanceOf` using function `one_msg`, defined in its own library, on Lines 23 and 27. In fact, most non-trivial Scilla programs are written in pure functional fragments. The pure fragment of Scilla conforms to System F calculus [Gir72] which has been extended with primitive data types, primitive operations, and user-defined algebraic

```
scilla_version 0
library FungibleToken
let min_int : Uint128 → Uint128 → Uint128 = (* ... *)
let le_int  : Uint128 → Uint128 → Bool    = (* ... *)
let one_msg : Msg → List Msg =
    (* Return singleton List with a message *)

contract FungibleToken
(owner : ByStr20, total_tokens : Uint128,
    decimals : Uint32, name : String, symbol : String)
field balances : Map ByStr20 Uint128 =
    let m = Emp ByStr20 Uint128 in
    builtin put m owner total_tokens
field allowed : Map ByStr20 (Map ByStr20 Uint128) =
    Emp ByStr20 (Map ByStr20 Uint128)

transition BalanceOf (tokenOwner : ByStr20)
  bal ← balances[tokenOwner];
  match bal with
  | Some v ⇒
    msg = {_tag : "BalanceOfResponse"; _recipient : _sender;
           address : tokenOwner; balance : v};
    msgs = one_msg msg; send msgs
  | None ⇒
    msg = {_tag : "BalanceOfResponse"; _recipient : _sender;
           address : tokenOwner; balance : zero};
    msgs = one_msg msg; send msgs
  end
end
transition TotalSupply ()
    (* code omitted *) end
transition Transfer (to : ByStr20, tokens : Uint128)
    (* code omitted *) end
transition TransferFrom (from : ByStr20, to : ByStr20,
                         tokens : Uint128)
    (* code omitted *) end
transition Approve (spender : ByStr20, tokens : Uint128)
    (* code omitted *) end
transition Allowance (tokenOwner : ByStr20, spender : ByStr20)
    (* code omitted *) end
```

Figure 2.4: Implementation of the `FungibleToken` contract in Scilla.

datatypes. While Scilla does not allow users to write recursive programs, it gives access to a few polymorphic recursive folds guaranteed to always terminate [Ser+19]. On the other hand, the imperative logic of transitions contains only primitive state-manipulating logic like assigning to contract fields [Ser+19].

2.4.1 Scilla's Monadic Interpreter

Scilla's semantics are implemented with a *monadic definitional interpreter* [SKH18]. In Section 2.1, we discussed how semantics of a language can be

```
1  let rec exp_eval (e, loc) env = match e with
2    let open EvalMonad.Let_syntax in
3    | Literal l → return (l, env)
4    | Var i →
5        let%bind v = Env.lookup env i in return (v, env)
6    | Let (i, _, lhs, rhs) →
7        let%bind lval, _ = exp_eval lhs env (e, U) in
8        let env' = Env.bind env (get_id i) lval in
9        let thunk () = exp_eval rhs env' in
10       wrap_eval thunk (e, E lval)
11   | GasExpr (g, e') →
12       let thunk () = exp_eval e' env in
13       let%bind cost = eval_gas_charge env g in
14       checkwrap_op thunk (Uint64.of_int cost)
15         ("Insufficient gas")
16   | ...
```

Figure 2.5: Snippet of Scilla's Expression Interpreter

defined using a definitional interpreter. In Section 2.3, we discussed how monads can be used to redefine a programming language's semantics. A monadic definitional interpreter implies that the interpreter's evaluate function returns a result type wrapped in a monad type in contrast to returning just the result type.

For Scilla's interpreter, the monad serves its key purpose of tracking resource consumption. Each interaction with a contract or deployment of a contract requires the emitter to pay a specific amount of *gas*. If the user's remaining amount of gas does not cover the cost of invoked executions, the interpreter terminates with an *out-of-gas* error [Ser+19]. This is done using the implemented *state*-monad to check whether the execution is successful so far by tracking intermediate execution states.

An example of monadic evaluation can be seen in the expression evaluate function in Figure 2.5. The figure shows evaluation of some of Scilla's pure components, namely evaluation of literals, variables, `let`-bindings, and gas expressions. At Line 2, the definition of the monad is imported along with its two standard operations `return` and bind which is encoded with the `let%bind` notation.

The defined monad data type used by the interpreter can be seen here:

```
type ('a,'b) result = Ok of 'a | Error of 'b
```

```
type nonrec ('a, 'b, 'c) t = (('a, 'b) result -> 'c) -> 'c
```

which we find to be a continuation-passing style (CPS) monad. As a result of the monad being in CPS, it does not explicitly declare the gas component in its type definition. The gas-tracking component `Gas` is later inferred as the monad is used as a gas-aware monad, expanding the polymorphic type `'c` to `Gas` →`'d` — a well-known technique for layering monads [Fil99]. It is exactly this set up that then allows us to extend the monad to keep track of the program's concrete semantics, which is discussed in Chapter 3.

It is worth noting that Scilla is not in full CPS, as that would require re-serialisation of closures [Ser+19]. To be in full CPS, continuations are always passed as an argument at every recursive call. However, the interpreter contains components that "cut" the CPS execution to intermittently check whether the user's allotted amount of gas covers their future computations. To "cut" the CPS execution, in this case, implies explicitly handling the continuation parameter of a CPS computation, usually done with the purpose of implementing some logic to determine whether or not to pass the continuation. Scilla gas accounting is enabled by pre-processing Scilla programs and wrapping every expression in a `GasExpr` which details the cost of computing the given expression. When the evaluator encounters `GasExpr` (as seen on Lines 11-15 of Figure 2.5), the interpreter calls `checkwrap` which checks whether there is enough gas resources for future computations, then decreasing the remaining amount of gas or propagating an error message. The implementation of `checkwrap` can be seen here:

```
(* Gas accounting *)
let checkwrap_op op_thunk cost emsg k remaining_gas =
  if remaining_gas > cost then
    op_thunk () k (remaining_gas - cost)
  else k (Error emsg) remaining_gas
```

Finally, Line 10 calls function `wrap_eval` which virtualises the recursive

call to collect specific runtime data but is ignored during evaluation, discussed in the next chapter.

3 Embedding the Harness for Collecting Semantics

As we have seen in Section 2.2, the result of static analyses is typically defined as an over-approximation of a program's concrete semantics. Therefore, in order to test the soundness of these analyses, we implement an additional harness for the definitional interpreter to collect concrete semantics, ideally, without altering the interpreter's logic. In this section, we introduce the process of embedding the collecting semantics monad into Scilla's interpreter for a simple collection of trace-semantics. The detailed account of more complex dynamic collecting is then described in the context of the case studies in Chapter 4.

3.1 Modularity of Semantics Collection

The original interpreter's structure streamlines the process of modularly introducing the collecting harness which is done by extending the existing CPS monad. Let us recall that, given the CPS monadic result type

```
type nonrec ('a, 'b, 'c) t = (('a, 'b) result -> 'c) -> 'c
```

the interpreter's designers are able to incorporate gas-tracking semantics by fine-tuning the abstract type `'c` to be `uint64` $\rightarrow$`'d`. Here, `uint64` is an OCaml 64-bit integer used to represent the allotted amount of gas for future computations, and `'d` is another abstract type. Instrumenting the interpreter with the semantics collecting harness then follows the

implementation of tracking gas consumption.

Let us denote `CollectedStates` to be a data type that stores intermediate execution states. We then extend the abstract type `'c` to expect gas and `CollectedStates` arguments. In other words, we aim to refine the continuation's return type to be `uint64 →CollectedStates →'d`, thus adding the semantic collecting component.

Similarly to how gas is handled, we need a procedure that appends new traces of states onto `CollectedStates`. The logic of what information is collected to grow the intermediary `CollectedStates` is contained within method `wrap_eval`. As we have seen in Figure 2.5, when evaluating the `Let`-expression, we "cut" the CPS execution by creating a closure `thunk` and explicitly updating the `CollectedStates` data type in the function `wrap_eval`. Once updated, we evaluate `thunk` to resume the evaluation of the program. The implementation of `wrap_eval`, shown below, shows the implementation of the method, where `update_seman` traverses the expression being evaluated and extracts and appends appropriate information.

```
let wrap_eval thunk collected_seman k remaining_gas
    current_seman_collect =
    thunk () k remaining_gas
        (update_seman current_seman_collect collected_seman)
```

A simple example of using the harness is extracting the footprint of expression evaluations, i.e., a chronological list of expressions that the interpreter evaluates. Given a program with a simple control flow below:

```
let x = Int32 42 in
let f = fun (z : Int) => x in
let y = x in
let a = y in
a
```

the resulting collected trace, when pretty-printed, is:

```
Let: x <- (Lit (Int32 42)) = ((Int32 42))
Fun: Var z -> (Variable x)
Let: f <- (Fun: Var (z) Body: Variable x) = (<closure>)
Variable: x -> (Int32 42)
Let: y <- (Variable x) = (Int32 42)
Variable: y -> (Int32 42)
Let: a <- (Variable y) = (Int32 42)
Variable: a -> (Int32 42)
```

In the trace, we can see how a `let`-binding is evaluated to bind some literal value to a variable, how an anonymous function is evaluated into a closure, and how a variable is evaluated into a literal it is bound to in the environment. Given the collected data, one can manipulate it to give more interesting accounts of data flow such as what values or variables flow into which variable definitions. The results of said manipulations can be seen below:

```
Variable x -> ( Lit (Int32 42) )
Variable f -> ( Fun: Var (z) Body: Variable x )
Variable y -> ( Variable x <- Lit (Int32 42) )
Variable a -> ( Variable y <- Variable x <- Lit (Int32 42))
```

This harness can then be extended to collect more advanced information such as the types that flow into entities or the data that flows into parameters. The extensions of the harness are detailed in later sections discussing performed case studies.

While we show only a small example of how intermediary execution states are collected in the pure fragment of Scilla, the addition of `wrap_eval` is propagated throughout the whole Scilla language. This includes the recording evaluation of contract parameters, mutable fields, transition and procedure parameters, as well as the evaluations of impure statements within. Although the idea is simple, the majority of effort is put into retrofitting every element of the language with care to make sure all necessary information is collected soundly. It is of utmost importance that the collected data is truthful and sound to begin with, as it is later used to test static results.

4 Specialised Collection and Case Studies

In the previous section, we outline the implementation of the general framework for dynamic semantic collection. Therefore, we now explore how to populate and refine our collected data to test specific static program analyses. In this section, we offer how the collected semantics are used to test static type conformance and the static type-flow analyses implemented in the Scilla to LLVM compiler.

4.1 Testing Type Conformance

Following the well-known mantra "well-typed programs do not go wrong", Scilla designers equipped the interpreter with a static type-checker to ensure type soundness. The definition of type soundness in Scilla states that "given sufficient amount of gas, for a well-typed term `e`, `e` should evaluate to a value `v` of the same type without error" [Ser+19; Hoa+22]. In other words, given enough gas for future executions, a term of some inferred type, say `T`, should successfully evaluate to a value of the same type `T`. Beyond the pure fragment of Scilla, when executing imperative programs, it is important to check that intermediate values and function parameters with a declared type get assigned values of that exact type. Given a type-driven compiler, the violation in the mentioned property can lead to a runtime error.

4.1.1 Collection of Type Flows into Identifiers

Before diving into how we collect type flows, it is important to make the distinction between *pure* and *impure* Scilla and how they are typed. The *pure* fragment of Scilla refers to the expressions, whereas the *impure* fragment of Scilla refers to the imperative fragment consisting of statements, fields, contract and transition parameters. As expressions produce a value, all expressions are typed with a type that describes what values can be produced. For example, an expression such as

```
let x = Int32 42 in
let f = fun (z: Int32) => z in
f x
```

has type `Int32`, which is a 32-bit integer type in Scilla. Therefore, when testing type-conformance for the pure fragment of Scilla, we ensure that all types inferred by the type-checker are consistent with run-time types of expressions. A statement, however, does not produce a value and, therefore, cannot have a type. As such, when checking type conformance for the impure segment of Scilla, we simply check whether a typed identifier such as a mutable field or a parameter received a value of its corresponding type.

Scilla programs come in either type-annotated (after type-checking) or unannotated (when evaluating) forms. In other words, when performing type-checking, a type-unannotated program is passed to the type-checker. The type-checker then returns the program with type annotations and whether or not is type-checks. The goal of the case study is to ensure that the type annotations of identifiers are consistent with their recorded run-time types. This is done by checking if the run-time flow of literals (which are typed) into an identifier conforms with its respective declared or inferred type. In other words, given a variable, a parameter, or a field is annotated to be of type `T` by the type-checker, we check that only terms

of type `T` flow into it.

Collection of Type Flows of Pure Scilla

To better understand how to collect the information we need to test type-conformance, let us properly define what it means for a type to flow into an identifier: a type `T` is considered to *flow* into an identifier when the evaluator binds the evaluated term `e` of type `T` to the identifier in the environment. For example, a type `T` flows into a function parameter when the function is applied to an expression `e` of type `T`.

Let us discuss the collection of type flows in the pure fragment of Scilla first. It is important to note that if an identifier has an explicitly declared type, we record it as well. When evaluating expressions, only `let`-expressions, function applications, and `match`-expressions update the environment by binding identifiers to values in the environment.

In the `let`-expressions, recording what type flows into an identifier is straight forward. Given a `let`-expression `let a = Int32 42`, we record how the type `Int32` flows into an identifier `a`.

In the `match`-expressions, we consider how a value is pattern-matched and then traverse its type to find what type flows into the declared variable. Consider the following `match`-expression:

```
let x = Pair {Bool Int32} True 2 in
match x with
| Pair a b => a
end
```

Knowing that `x` is of the type `Pair {Bool Int32}`, one can traverse its type and check what it is pattern matched to to record how the type `Bool` flows into the identifier `a` and the type `Int32` flows into identifier `b`.

Given function declaration and function application expressions below:

```
let f = fun (x : Int32) => x in
let a = Int32 1 in
f a
```

since a is passed to parameter x, it is intuitive that the type Int32 flows into the parameter x. The collection procedure, however, is a little more intricate. In Scilla, anonymous functions are evaluated to be OCaml closures that expect Scilla arguments. Therefore, when a function application is evaluated, the OCaml closure is applied to the respective Scilla arguments. Since the function is stored as a closure in the environment, at the point of evaluating the function application, we no longer have access to the parameters of the function to record what values flow into which parameters.

To solve this, we incorporate the procedure of recording what is passed to the function parameters into the closure as well. In other words, when a function is evaluated, it is stored as a closure that records the argument's types that flow into the function parameters before evaluating the body of the function given the said arguments.

Having detailed how type flow collection is recorded for expressions, let us look at an example below.

```
let x = Int32 42 in
let f = fun (z : Int32) =>
    let b = x in
    fun (c : Int32) => z
in
let a = Int32 1 in
let d = Int32 2 in
f a d
```

This program is taken from the collection of test programs written by the Scilla designers. Evaluating said program with the harnessed evaluator would return a trace seen in Figure 4.1.

Collection of Type Flows of Impure Scilla

Having described our approach of collecting type flows for the pure fragment of Scilla, we move onto handling the impure fragments containing statements, transitions, procedures, fields, and contract definitions. As most statements in Scilla are either load or store statements that

```
Variable x: Int32 <- (Lit (Int32 42): Int32)
Variable f: Int32 <- (Fun (Var z: Int32): Int32)
Variable a: Int32 <- (Lit (Int32 1): Int32)
Variable d: Int32 <- (Lit (Int32 2): Int32)
Variable z: Int32 <- (Variable a: Int32) <- (Lit (Int32 1): Int32)
Variable b: Int32 <- (Variable x: Int32) <- (Lit (Int32 42): Int32)
Variable c: Int32 <- (Variable d: Int32) <- (Lit (Int32 2): Int32)
```

Figure 4.1: Trace of Run-Time Type Flows

bind a value to a newly declared variable, similarly to handling the `let`-expressions, we simply record the type of the value being stored in an identifier. When a transition, procedure, or contract are applied to some arguments, we simply record the parameter's type and what type flows into said parameter. The collection of the types that flow into a map identifier is a little less straightforward as it requires traversal of the map type. Consider an example where, given a map `m` of type

```
Map ByStr20 (Map ByStr20 Int128)
```

a map-update statement such as

```
m[k1][k2] := v
```

is clearly updating the value of the type `Int128` where `v` should of the type `Int128`, `k1` and `k2` should be of the type `ByStr20`. Here, `Int128` is a Scilla type for 128 bit integers, and `ByStr20` is a Scilla type for a hexadecimal Byte String representing a 20 byte address. When collecting information about this statement, we traverse the map's type to find what type is the identifier `m[k1][k2]` (`Int128`), and then record how variable `v`'s type flows into the identifier as well. The resulting collected trace looks similar to the one seen in Figure 4.1.

Side Note on Name Shadowing

During the implementation of the collection harness, an issue occurred to us when we attempted to organise the collected information. When

collecting the types that flow into the identifiers, we were only identifying the identifiers by their name. In other words, for a program such as

```
let b = Int32 42 in
let b = True in
b
```

we would collect how both types `Int32` and `Bool` flow into some identifier `b`, which clearly violates type conformance. Although the issue was thought of, it did not come up when testing against the collection of user-written or randomly generated Scilla programs (Section 4.1.2). In fact, all user-written programs steer clear of reusing the same name for variables, fields, or parameters, and randomly generated programs always come up with a new name for any newly declared identifier. Therefore, for the program above, it is sufficient for us to simply overwrite type flows of redundant variable definitions. In other words, if a declared variable hasn't been used before it is re-declared with a new definition, we overwrite the previous type flow, no longer considered live, with the new one.

In the future implementations, however, it is best to not overwrite any type flows to honor the completeness of type flow collection. Instead, one could traverse the Scilla program before the evaluation and tag different variables with the location they were defined at. By doing so, given the following program

```
1 let b = Int32 2 in
2 let f = fun (x: Int32) => x in
3 let z = f b in
4 let b = True in
5 b
```

our collection results will be able to differentiate that the variable `b` used on Line 3 is the one defined on Line 1, whereas the variable `b` seen on Line 5 is the one defined on Line 4.

4.1.2 Results of Testing Type Conformance

Our strategy for testing the type conformance, and therefore, Scilla's static type-checker, involves confirming that the inferred type of an identifier is indeed accurate to what is seen in run-time. In essence, we check that given an identifier `x` of type `T`, i.e., `x: T`, only values of type `T` flow into it. It is important that we make the following clarification to our type conformance definition: if some type `U` is a sub-type of type `T`, it is still safe to assign a value of type `U` to an identifier of type `T`. However, assigning a value of type `T` to a variable of type `U` is not safe. It is also important to note that the type conformance is checked with respect to *runtime types*. In other words, all type variables are already instantiated with ground types when checked.

The testing framework works as following:

- Given a Scilla program, we first run the type-checker to find all of the inferred types of the identifiers.
- We evaluate the Scilla program using our evaluator with the collecting harness to collect all of the types that flows into all of the identifiers.
- We then check whether the types flowing into said identifiers are assignable to their inferred type.

The framework was first run on 105 Scilla "good" test programs written by designers of the language [Hoa21]. Since all the programs are human written and considered to be "good", i.e., soundly written, no bugs were found when evaluating said programs. However, this is to be expected as testing on a few well-written programs does not not yield substantial results.

In addition to testing using human written programs, the harness was able to test type conformance on randomly generated programs. As part

of this project, we had an opportunity to work along with researchers who implemented random generation of Scilla programs using QuickChick. As part of their project, the researchers were able to efficiently generate well-typed System F programs along with imperative state-manipulating code to create full-blown Scilla smart contracts [Hoa+22]. Testing against randomly generated programs not only allows the harness to test against more programs, but also against a wider variety of programs with potentially more interesting type flows.

When testing type conformance against these randomly generated Scilla contracts, we found an unusual bug in the type-checker. The harness discovered how values of sub-types of `ByteString` are implicitly up-cast to `ByteString`. Scilla type system details the address sub-typing hierarchy where an address type `ByStr20 with ... end` is a sub-type of `ByStr20`, an address type `ByStr20 with contract ... end` is a sub-type of `ByStr20 with end`, and other such relations between address types. Our harness was able to detect how variables with the sub-types of `ByStr20`, such as `ByStr with end`, are implicitly up-cast to `ByStr20`. Assigning a value of the type `ByStr20` to an identifier of the type `ByStr20 with end` should be considered unsound and caught by the type-checker. However, since the identifier's type is up-cast to `ByStr20`, such error is not noted. While this bug does not compromise safety guarantees of contract execution, it might affect the correctness of the compiler.

4.2 Testing the Type-Flow Analysis

One great property that the compiler from Scilla to LLVM takes advantage of is when a smart contract is deployed, all libraries it might potentially need are known at compile time. This allows for a whole-program optimisation such as full monomorphisation of polymorphic definitions

[Nag+20]. Scilla features polymorphism as it is considered one of the linchpins of modern typed functional languages which, unfortunately, comes with a real performance penalty [EP17]. One of the key tools to combating the performance penalty is monomorphisation where instead of evaluating the polymorphic terms, the program generates specialised implementations of said terms instantiated with necessary ground types. To find out which ground types are necessary for the functions to be instantiated with, the compiler performs a carefully designed *type-flow analysis* that conservatively determines which types variables are instantiated with which ground types [Wee06; Nag+20].

The outcome to the type-flow analysis is the collection of all type variables in the program paired with all ground types it might be instantiated with. As discussed in Section 2.2, the type-flow analysis safely approximates which ground types a type variable may be instantiated with. Therefore, we can test the analysis using our collecting interpreter to check whether the type variables are indeed instantiated with predicted ground types by evaluating the program. For example, consider the following Scilla program

```
let id = tfun 'X => fun (x: 'X) => x in
let idint = @id Int32 in
let a = Int32 42 in
let idstring = @id String in
let s = "hello world!" in
idstring s
```

we can see that the polymorphic function `id` is instantiated with the types `Int32` and `String` before being applied to a value of the type `Int32` and a value of the type `String` respectively. By evaluating the program, we can dynamically collect how the types applied to the type parameter `'X` are indeed `Int32` and `String`. Recalling that the static analyses are allowed to over-approximate results, if the analyses predicts that `'X` would be instantiated with ground types `Int32`, `String`, and `Bool`, albeit a little odd, it would still be a valid result. However, we do not accept under-

approximated analyses results. For example, if the analysis predicts that only `Int32` instantiates `'X`, it would not be considered a sound result.

While the Scilla to LLVM compiler has a few different optimisations, we decided to test the monomorphisation pass because (1) the type-flow analysis is one of the more complex and intricate analysis implemented in the compiler, making it less trivial to test and more suitable for an interesting case study, and (2) a buggy and imprecise type-flow analysis implies initialisation of polymorphic functions with types that are not used, generating redundant functions and thus compromising the idea of "optimisations", or not initialising with a type that is used which can lead to run-time errors.

4.2.1 Collection of Type Flows into Type Variables

The methodology for collecting which ground types flow into which type variables is akin to the method of collecting the type flows into identifiers discussed in Section 4.1.1. Instead of looking for places where a value is bound to an identifier, we look for the code where a type variable is substituted, namely at type applications.

Similarly to functions, when the interpreter evaluates Scilla type functions, it creates an OCaml closure that expects a Scilla ground type as an argument before substituting the appropriate type variable for the ground type into the body of the function. The closure is then stored in the environment. Therefore, when we evaluate type applications, we no longer have access to the type function parameter to record which ground type flows into the type parameter. As such, we incorporate the collection procedure into the type function closure as well. The closure then contains the process of recording what ground types are applied to which type parameters, before substituting the parameters with the ground types and evaluating the function body.

4.2.2 Results of Testing the Type-Flow Analysis

As mentioned above, the aim is to check the soundness of type-flow analysis by checking whether the predicted set of ground types to instantiate a type variable is a super-set of the actual set of runtime ground types instantiating the type variable. The testing framework works as follows:

- Given a Scilla program, we first run the type-flow analysis compiler pass to find the predicted set of types instantiating a type variable.
- We run the evaluator with the harness to collect all types applied to the type variable.
- We check whether the later set is a subset of the former set.

Similarly to the testing type conformance, we first tested against the user-written "good" test programs in the Scilla repository[1]. While running the test harness did not reveal much, it acted as a sanity check for the harness.

More interesting results arose when we tested the harness against randomly generated Scilla programs. We found that type-flow analysis to be incredibly robust as no bugs were found in the process of testing it. In fact, our tests revealed that the analysis' predicted set of ground types is the exact set dynamically collected for all randomly generated programs. This precision, however, can be explained by the fact that all type abstractions from the randomly generated programs were instantiated exactly once. Ideally, if the random program generator generated programs with type abstractions instantiated more than once or even type abstractions within type abstractions, there could be a greater guarantee of the possibility of finding implementation bugs. However, this can be part of the future work of this project.

[1]https://github.com/Zilliqa/scilla

Only in the cases when type application occurred in different branches of `match`-expressions did the analysis return expected over-approximation of dynamically collected ground types. For example, given a program such as

```
let f = tfun 'X => fun (x: 'X) => x in
let a = True in
match a with
| True => let fstring = @f String in f "abc"
| False => let fbool = @f Bool in f False
end
```

the type-flow analysis will account for both branches of the `match`-expression and predict that the type variable `'X` might be instantiated with either types `String` or `Bool`. Collecting what types `'X` is instantiated with in concrete execution, of course, returns that `'X` is only instantiated with type `String`. As these over-approximations are valid, our harness found no bugs in these test cases.

5 Related Work

Researchers have looked into many techniques of testing soundness of static analyses with a wide spectrum of degree of human effort. Some have written *manual proofs* to prove an analysis' soundness, as shown by Midtgaard et al. who proved the soundness of Shivers' methodology of analysing a program's control-flow [MAM12]. Similarly, Blazy et al. then contributed to the body of scholarship by performing *formal verification* using the Coq proof assistant to prove soundness of the results of a static analyser [Bla+13]. Both methods require an incredibly high degree of human effort but can ensure the absolute absence of program bugs.

Beyond formalising proofs, testing properties of static analyses by generating wide spectrum of programs, as done in this thesis, is, of course, not novel. Bugariu et al. presented an automatic technique to test soundness and precision of abstract domains by generating random test programs using gray-box fuzzing [Bug+18]. Similarly, Midtgaard and Møller employed QuickCheck to validate algebraic properties of abstract domains through generation of random abstract state components [MM17]. Klinger et al. and Taneja et al. evaluated different analyzers' soundness and precision with randomly generated programs, where the former used differential testing between analyzers, and the later computed sound and maximally precise programs using SMT solvers before comparing them to static analyser's results [KCW19; TLR20].

Additionally, as briefly mentioned in Chapter 2, implementing collecting semantics in a monad as part of an evaluator has been explored as well [Dar+17; Ser+13]. These implementations, however, have only been implemented for toy functional languages and outside of the context of OCaml. Furthermore, these approaches were then only used to redefine a program's semantics or abstract a language's semantics.

The novelty of our work lies in adopting the idea of monadic collecting semantics in a definitional interpreter to *test* static analyses of higher-order languages.

6 Discussion

6.1 Extensibility and Future Work

The aim of this project was to refactor a production-scale definitional interpreter to contain a state-collecting harness to test static type-conformance and static type-flow analysis. For our future work, as mentioned in Section 4.2.2, the probability of finding an implementation bug would benefit is the randomly generated programs had multiple instantiations of a type abstraction or nested type abstractions. Additionally, we could extend our harness to test other existing static optimisations in the Scilla to LLVM compiler such as dead-code elimination, uncurrying analysis, early evaluations of library functions [Nag+20], or even potential analyses that are not yet implemented such as just in time compilation or loop optimisations. This extensibility comes from the straightforwardness of specialising what data can be dynamically collected.

Another tool that could be interesting to test is *CoSplit* [PKS21]. *CoSplit* is a static program analysis tool that infers smart contract properties such as the ownership and commutativity summaries, which are then used to maximise parallelism. To infer these properties, CoSplit performs static analysis on smart contracts written in Scilla, which we can test with our harness.

To discuss the extensibility of the project's idea to other languages,

it is important to recall the context in which this project was performed in. Scilla provided us with many infrastructures upon which we piggy-backed on such as a definitional interpreter equipped with a CPS monad and, most significantly, a random Scilla program generator. Without randomly generated programs, the analysis might not have been as fruitful and extensive. However, given a developer has a definitional interpreter they are familiar with, it suffices to refactor it to become monadic (without CPS) to allow for dynamic state collection [Hoa21]. This enhanced definitional interpreter then becomes a useful debugging tool for testing new language features or its corresponding static analyses.

6.2 Conclusion

In this project, we present a detailed account of embedding an state-collecting harness into a definitional interpreter to test static program analyses. We start by providing a thorough study of technical terms heavily relied on in the project's report (Chapter 2). We then transform the definitional interpreter's CPS monad to take into account the dynamic state collection (Chapter 3). Finally, we described two case studies of testing static type conformance and static type-flow analysis by specialising collected properties, with which we evaluate the static properties through user-written and randomly generator Scilla programs (Chapter 4).

Our two cases studies revealed interesting details about the software we were testing. Firstly, testing type conformance revealed an odd bug of implicit up-casting of an address type performed by the type-checker. While the bug does not compromise safety guarantees of contract execution, it affects the correctness of the compiler. This newly found bug has been disclosed to the Scilla developers, and has been fixed in the updated version of Scilla. Secondly, testing the type-flow analysis

revealed the immaculate precision of the analysis, although the precision could be due to how types are instantiated in the randomly generated Scilla programs.

While our methodology does not prove complete lack of bugs, we have demonstrated how this simple technique can be a stepping stone to a compiler developer's bug-free dreams.

Bibliography

[AC76] Frances E. Allen and John Cocke. "A program data flow analysis procedure". In: *Communications of the ACM* 19.3 (1976), p. 137 (cit. on pp. vi, 10–12).

[Age+03] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. "A functional correspondence between evaluators and abstract machines". In: *Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*. 2003, pp. 8–19 (cit. on p. 8).

[AR17] Nada Amin and Tiark Rompf. "Type soundness proofs with definitional interpreters". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. 2017, pp. 666–679 (cit. on pp. 2, 6).

[BHM00] Nick Benton, John Hughes, and Eugenio Moggi. "Monads and effects". In: *International Summer School on Applied Semantics*. Springer. 2000, pp. 42–122 (cit. on pp. vii, 2, 14).

[Bla+13] Sandrine Blazy, Vincent Laporte, André Maroneze, and David Pichardie. "Formal verification of a C value analysis based on abstract interpretation". In: *International Static Analysis Symposium*. Springer. 2013, pp. 324–344 (cit. on p. 35).

[Bla+19] Sam Blackshear, Evan Cheng, David L Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Dario

Russi Rain, Stephane Sezer, et al. “Move: A language with programmable resources”. In: *Libra Assoc* (2019) (cit. on p. 1).

[Bug+18] Alexandra Bugariu, Valentin Wüstholz, Maria Christakis, and Peter Müller. “Automatically testing implementations of numerical abstract domains”. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 2018, pp. 768–778 (cit. on pp. vi, 8, 35).

[Dar+17] David Darais, Nicholas Labich, Phuc C Nguyen, and David Van Horn. “Abstracting definitional interpreters (functional pearl)”. In: *Proceedings of the ACM on Programming Languages* 1.ICFP (2017), pp. 1–25 (cit. on pp. vi, 1, 3, 8, 36).

[DF92] Oliver Danvy and Andrzex Filinski. “Representing control: A study of the CPS transformation”. In: *Mathematical structures in computer science* 2.4 (1992), pp. 361–391 (cit. on p. 7).

[DN01] Olivier Danvy and Lasse R Nielsen. “Defunctionalization at work”. In: *Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming*. 2001, pp. 162–174 (cit. on pp. 6, 7).

[EP17] Richard A Eisenberg and Simon Peyton Jones. “Levity polymorphism”. In: *ACM SIGPLAN Notices* 52.6 (2017), pp. 525–539 (cit. on pp. 1, 31).

[Fil94] Andrzej Filinski. “Representing monads”. In: *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1994, pp. 446–457 (cit. on p. 2).

[Fil99] Andrzej Filinski. “Representing layered monads”. In: *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1999, pp. 175–188 (cit. on p. 18).

[Flo93] Robert W Floyd. "Assigning meanings to programs". In: *Program Verification*. Springer, 1993, pp. 65–81 (cit. on p. 5).

[Gir72] Jean-Yves Girard. "Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur". Thèse d'État. Paris, France: Université de Paris VII, 1972 (cit. on p. 15).

[Hoa+22] Tram Hoang, Anton Trunov, Leonidas Lampropoulos, and Ilya Sergey. "Random testing of a higher-order blockchain language (experience report)". In: *Proceedings of the ACM on Programming Languages* 6.ICFP (2022), pp. 886–901 (cit. on pp. 1, 15, 23, 30).

[Hoa21] Tram Hoang. "Testing Static Code Analysis with Monadic Definitional Interpreter". Bachelor's Thesis. 2021 (cit. on pp. 29, 38).

[KCW19] Christian Klinger, Maria Christakis, and Valentin Wüstholz. "Differentially testing soundness and precision of program analyzers". In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2019, pp. 239–250 (cit. on p. 35).

[KRS94] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. "Partial dead code elimination". In: *ACM Sigplan Notices* 29.6 (1994), pp. 147–158 (cit. on p. 1).

[KSS17] Uday Khedker, Amitabha Sanyal, and Bageshri Sathe. *Data flow analysis: theory and practice*. CRC Press, 2017 (cit. on pp. 11, 12).

[LHJ95] Sheng Liang, Paul Hudak, and Mark Jones. "Monad transformers and modular interpreters". In: *Proceedings of the 22nd*

ACM SIGPLAN-SIGACT symposium on Principles of programming languages. 1995, pp. 333–343 (cit. on pp. vii, 3).

[MAM12] Jan Midtgaard, Michael D Adams, and Matthew Might. "A structural soundness proof for Shivers's escape technique". In: *International Static Analysis Symposium*. Springer. 2012, pp. 352–369 (cit. on p. 35).

[Mig10] Matthew Might. "Abstract interpreters for free". In: *International Static Analysis Symposium*. Springer. 2010, pp. 407–421 (cit. on pp. vi, 8).

[MM17] Jan Midtgaard and Anders Møller. "Quickchecking static analysis properties". In: *Software Testing, Verification and Reliability* 27.6 (2017), e1640 (cit. on pp. 14, 35).

[MRL13] Jan Midtgaard, Norman Ramsey, and Bradford Larsen. "Engineering definitional interpreters". In: *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming*. 2013, pp. 121–132 (cit. on pp. 2, 8).

[Nag+20] Vaivaswatha Nagaraj, Jacob Johannsen, Anton Trunov, George Pırlea, Amrit Kumar, and Ilya Sergey. "Compiling a Higher-Order Smart Contract Language to LLVM". In: *arXiv preprint arXiv:2008.05555* (2020) (cit. on pp. vii, 1, 3, 31, 37).

[NNH04] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of program analysis*. Springer Science & Business Media, 2004 (cit. on pp. vi, 2, 9–13).

[Pał+11] Michał H Pałka, Koen Claessen, Alejandro Russo, and John Hughes. "Testing an optimising compiler by generating random lambda terms". In: *Proceedings of the 6th International Workshop on Automation of Software Test*. 2011, pp. 91–97 (cit. on p. 1).

[Pie02] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002 (cit. on p. 6).

[PKS21] George Pırlea, Amrit Kumar, and Ilya Sergey. "Practical smart contract sharding with ownership and commutativity analysis". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2021, pp. 1327–1341 (cit. on p. 37).

[Rey72] John C. Reynolds. "Definitional Interpreters for Higher-Order Programming Languages". In: *Proceedings of 25th ACM National Conference*. Reprinted in Higher-Order and Symbolic Computation 11(4):363-397, 1998, with a foreword [Rey98a]. Boston, Massachusetts, 1972, pp. 717–740 (cit. on pp. 2, 5–7).

[Rey98a] John C Reynolds. "Definitional interpreters for higher-order programming languages". In: *Higher-order and symbolic computation* 11.4 (1998), pp. 363–397 (cit. on p. 44).

[Rey98b] John C Reynolds. "Definitional interpreters revisited". In: *Higher-Order and Symbolic Computation* 11.4 (1998), pp. 355–361 (cit. on pp. vi, 2, 6).

[Ser+13] Ilya Sergey, Dominique Devriese, Matthew Might, Jan Midtgaard, David Darais, Dave Clarke, and Frank Piessens. "Monadic abstract interpreters". In: *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 2013, pp. 399–410 (cit. on pp. vii, 2, 14, 36).

[Ser+19] Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. "Safer smart contract programming with Scilla". In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (2019), pp. 1–30 (cit. on pp. vii, 1, 3, 15–18, 23).

[Shi91] Olin Shivers. “Control-Flow Analysis of Higher-Order Languages or Taming Lambda”. Technical Report CMU-CS-91-145. PhD thesis. Pittsburgh, Pennsylvania: School of Computer Science, Carnegie Mellon University, May 1991 (cit. on pp. vi, 1, 2, 8, 9, 11–13).

[SKH18] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. “Scilla: a smart contract intermediate-level language”. In: *arXiv preprint arXiv:1801.00687* (2018) (cit. on pp. vii, 1, 3, 15, 16).

[TLR20] Jubi Taneja, Zhengyang Liu, and John Regehr. “Testing static analyses for precision and soundness”. In: *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. 2020, pp. 81–93 (cit. on p. 35).

[Wee06] Stephen Weeks. “Whole-program compilation in MLton”. In: *ML* 6 (2006), pp. 1–1 (cit. on pp. 1, 31).