YaleNUSCollege

Testing Static Code Analyses with Monadic Definitional Interpreters

Hoang Ngoc Tram

Capstone Final Report for BSc (Honours) in
Mathematical, Computational and Statistical Sciences
Supervised by: Dr. Ilya Sergey
AY 2020/2021

Yale-NUS College Capstone Project

DECLARATION & CONSENT

1. I declare that the product of this Project, the Thesis, is the end result of my own work and that due acknowledgement has been given in the bibliography and references to ALL sources be they printed, electronic, or personal, in accordance with the academic regulations of Yale-NUS College.

2. I acknowledge that the Thesis is subject to the policies relating to Yale-NUS College Intellectual Property (Yale-NUS HR 039).

ACCESS LEVEL

3. I agree, in consultation with my supervisor(s), that the Thesis be given the access level specified below: [check one only]

✓ Unrestricted access
Make the Thesis immediately available for worldwide access.

o Access restricted to Yale-NUS College for a limited period
Make the Thesis immediately available for Yale-NUS College access only from _____ (mm/yyyy) to _____ (mm/yyyy), up to a maximum of 2 years for the following reason(s): (please specify; attach a separate sheet if necessary):
_____.

After this period, the Thesis will be made available for worldwide access.

o Other restrictions: (please specify if any part of your thesis should be restricted)
_____
_____

Hoang Ngoc Tram/ Cendana College
_____
Name & Residential College of Student

_____ 03.04.2021
Signature of Student Date

Prof. Ilya Sergey 03 April 2021
Name & Signature of Supervisor Date

Acknowledgements

I am forever grateful for the support of the following people who have shaped me and my experience at Yale-NUS College.

My capstone advisor Dr. Ilya Sergey, who has provided me with generous guidance, advice, and wisdom throughout the past 2 years of my college career. Without his patience and commitment to teaching, I might have never realised my passion for computer science.

My family and my puppy Pepsi for always welcoming me home.

The Yale-NUS women's basketball team for the basketball games and practices filled with giggles, camaraderie, and good memories.

Finally, my wonderful friends - Alysha, Sydney, Raya, Denise, Emma, and Gabe - for making the last 4 years unforgettable.

YALE-NUS COLLEGE

Abstract

B.Sc (Hons)

Testing Static Code Analyses
with Monadic Definitional Interpreters

by HOANG Ngoc Tram

Static code analyses, like any other software artifacts, are prone to having bugs. In this thesis, we describe the design and implementation of a refactoring technique, aimed at instrumenting definitional interpreters for testing static analyses. As modifying a definitional interpreter is the key component for dynamically collecting semantic information, our approach utilises monads — a foundational mechanism for implementing side effects in functional programming languages. We showcase the developed technique by applying it to a production-scale definitional interpreter (of a smart contract language used by a popular blockchain system) and employing it for testing the language's type safety claims.

Keywords: Static Analysis, Monads, Definitional Interpreter, Scilla

Contents

Chapter 1

Introduction

1.1 Motivation

Computer programs have bugs. Programming language researchers have devoted large amounts of effort to find these bugs automatically via a toolset of semantics-based code analyses such as type checking, effect analysis, control- and data-flow analyses and others [1, 4, 10, 16, 24].

However, program analysis tools, like any other software artifacts, also have bugs. The consequences of having bugs within the checker tools, inducing failures to find bugs, announcing non-existent bugs, and even silently introducing new bugs, can be disastrous. Example of such bugs include:

- missing the following type assignment error in a Java program

```
public int x;
x = "abc";
```

- misclassifying code as dead within static program analysis designed to discover dead code (code that, upon removal, does not affect the program's result [8]) (Figure 1.1)

This begs the question, who tests the checkers?

```
int c = 6
if (c > 5) {
  x = 42;
} else {
  x = 1; // True dead code
}
```

```
int c += 1;
if (c > 5) {
  x = 42;
} else {
  x = 1; // Not a dead code
}
```

FIGURE 1.1: Dead-Code Elimination Error

This project devises a testing technique that starts with using *definitional interpreters* [18]. The checkers, i.e., static analyses tools, give **abstract results** that over-approximate the run-time behaviour in order to account for non-determinism [10]. On the other hand, definitional interpreters are built to give a simple, clear, and readable account of program semantics, albeit sacrificing any semblance of efficiency [11, 18]. In other words, definitional interpreters output **concrete results** - the "ultimate truth" of the program's run-time behaviour. Thus, we can test the static analysis by checking if its abstract results agree with the definitional interpreter's concrete results.

1.2 Goals of this project

This project aims to provide a proof of concept of checking the static code analysis of a program with a definitional interpreter. Using dead-code elimination analyses (Figure 1.1) as an example, running the program through the definitional interpreter allows us to confirm whether the expression `(x=1)` is indeed dead. This is done by modifying the interpreter to dynamically record whether the line of code is evaluated or not. However, due to their concise and simple design, any additional modifications must not interfere with the interpreter's core logic.

1.3 Our Approach

This project makes use of *monads* to equip an interpreter with the capabilities of dynamic information collecting without changing the interpreter's core logic. Monads offer a near-complete reinterpretation of computations, *e.g.* the list monad grants computations with non-deterministic behaviour [21]. Since they were originally adapted to provide an abstraction for mutations in purely functional languages, monadically expressed semantics can make current computational effects explicit or fully hidden. Therefore, monads allows us to incorporate new features modularly, isolated from the core logic of the interpreter [9].

1.4 Challenges and Contributions

In our project, we revise a definitional interpreter for a new programming language for safe contracts called Scilla. The revision of Scilla's interpreter would then allow us to tests its respective static analyses. However, revising a production-scale definitional interpreter is not be trivial. Incorporating monadic semantics into Scilla's interpreter is, in itself, a challenge due to a number of design trade-offs discussed in the later chapters. Another challenge comes from the Scilla interpreter already being parameterised with a *continuation passing style* (CPS) [5] monad, requiring us to define a suitable notion of monadic semantics that would allow for static analyses testing.

This project is significant in the following aspects.

1. We devise a monad for collecting semantics [20] for System F [16].

2. We implement the monad into a production-scale definitional interpreter for the Scilla language [23].

3. We evaluate our implementation on a particular analysis, namely type checking.

1.5 Outline

The remainder of the report is structured as follows. Chapter 2 builds a foundational understanding of polymorphic lambda calculus (System F), accompanied by an implementation of a simple System F interpreter and type checker in OCaml. Chapter 3 describes the process of monadically parameterising our definitional interpreter to test static code analyses. Additionally, we discuss how we refactored our monad to be a CPS monad, thus avoiding running into call stack overflow. The final interpreter for system F contains the enhanced monad constructed for semantics accumulation. In chapter 4, we incorporate the ideas from the previous chapters into a real-world interpreter - the Scilla interpreter. More specifically, we describe the process and limitations of embedding the collecting semantics monad into a real-world interpreter. Chapter 5 describes our application of the enhanced interpreter to a case study, which tests the Scilla's static type checker. Finally, Chapter 6 concludes with a discussion and future work.

Chapter 2

Background

Before going to the industrial interpreter, we must hone our testing approach on a minimalistic foundational framework. As such, we choose to experiment with a *Polymorphic Lambda Calculus* (System F) interpreter. The following chapter introduces System F's influence within the programming language field, and highlights its syntax, rules, and properties that led to its popularity (Section 2.1). Afterwards, we describe our implementation of a simple System F definitional interpreter (Section 2.2).

2.1 Polymorphic Lambda Calculus

System F [6, 19] is used extensively as a research vehicle for foundational work on parametric polymorphism [16, 23]. Additionally, the framework has influenced and formed the basis of many languages, such as Haskell, OCaml, Scala, and, to some extent, Java, C# and others, all of which are supported by a large body of research on their static analyses and compilation techniques [16, 23]. Furthermore, we experiment with System F for its rich interactions between values and types, which is a perfect avenue for testing.

Syntax and Rules

`t` ::=		**(expressions...)**
	`x`	variable
	λ`x : T.t`	abstraction
	`t t`	application
	Λ`X.t`	type abstraction
	`t[T]`	type application
`v` ::=		**(values...)**
	λ`x : T.t`	abstraction value
	Λ`X.t`	type abstraction value
`T` ::=		**(types...)**
	`X`	type variable
	`T` $\rightarrow$ `T`	type of functions
	$\forall$`X.T`	universal type
Γ ::=		**(environment...)**
	$\emptyset$	empty context
	Γ`, x : T`	expression variable binding
	Γ`, X`	type variable binding

FIGURE 2.1: Syntax for System F

The definition of System F is an extension of *simply typed lambda calculus*, denoted as $\lambda_{\rightarrow}$ [16]. $\lambda_{\rightarrow}$ uses lambda **abstractions** to factor variables out of expressions, then instantiate said variables using **applications**. For example, given an expression `t`, we can abstract a variable `x` of type `T` from `t`, denoted as λ`x : T.t`. We can then apply some value `s` back into `t`, written as `(t s)`, where function `t` takes `s` as an argument [14].

In $\lambda_{\rightarrow}$, variables and expressions can only have base types (such as `Int` or `Bool`) or function types [14]. Function types have the shape `X` $\rightarrow$ `Y`, where `X` and `Y` can be either base types or function types.

System F extends $\lambda_{\rightarrow}$ by adding polymorphic types of shape $\forall$`X.t` [16]. Now, types can be abstracted from expressions in a similar manner to

variable abstractions. The syntax of abstracting a type parameter $\mathtt{X}$ from an expression $\mathtt{t}$ is $\Lambda\mathtt{X}.\mathtt{t}$, while applications are written as $\mathtt{t}[\mathtt{T}]$. Figure 2.1 shows the complete syntax of System F, written based on the *Types and Programming Languages* book [16].

Additionally, expressions must follow a set of evaluation and typing rules [16]. For instance, one of the evaluation rules, the *beta-reduction* rule, says that an application in the form of

$$(\lambda \mathtt{x} : \mathtt{T}.\mathtt{t}_{12})\mathtt{v}_2 \rightarrow \{\mathtt{x} \mapsto \mathtt{v}_2\}\mathtt{t}_{12} \qquad \text{(E-AppAbs)}$$

can be evaluated through substituting the term $\mathtt{x}$ with $\mathtt{v}_2$ within the expression $\mathtt{t}_{12}$. An example of a typing rule would be the *typing-judgement* rule for expression applications [16, 14].

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \rightarrow \mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1\mathtt{t}_2 : \mathtt{T}_{12}} \qquad \text{(T-APP)}$$

Let Γ denote the *environment*, ie. the key-value list, where the keys are the variables and the values are their respective types. The rule states that we must assert the following 2 premises

- the term $\mathtt{t}_1$ must have the type $\mathtt{T}_{11} \rightarrow \mathtt{T}_{12}$ in the environment Γ
- $\mathtt{t}_2$ has type $\mathtt{T}_{11}$ in the environment Γ

to be true to allow for the application of $\mathtt{t}_1$ on $\mathtt{t}_2$ with a resulting type of $\mathtt{T}_{12}$ in the environment Γ.

Given the rules, here's a simple example. Say t is a polymorphic `id` function $(\mathtt{id} = \Lambda\mathtt{X}.\ \lambda\mathtt{x} : \mathtt{X}.\ \mathtt{x})$ with a type of $(\mathtt{id} = \forall\mathtt{X}.\mathtt{X} \rightarrow \mathtt{X})$, then we can

apply type `Int` to get `((λ: Int.x): Int → Int)`. Applying some integer `0` as well would just return us `(0:Int)`.

2.2 A Definitional Interpreter for System F

The following section transitions from the mathematical notations of System F to its OCaml implementation as a definitional interpreter.

```
module SystemFOSignature = struct
  type var = string
  type tvar = string

  (*Types*)
  type ty =
    | TVar of tvar (* X *)
    | TFunc of ty * ty (* T -> T *)
    (*For all X, T*)
    | TForAll of tvar * ty
    | TInt

  (* Binary Operation *)
  type binop =
    | Add
    | Sub
    | Mul
    | Div

  (*Terms*)
  type exp =
    | Int of int
    | Var of var
    | ETVar of tvar
    | Abs of var * ty * exp
    | App of exp * exp
    | ETAbs of tvar * exp
    | ETApp of exp * exp
    | Typ of ty
    | Binop of binop * exp * exp

  type value =
    | IntV of int
    | Closure of environment *
        var option * ty * exp
    | TypV of ty
end
```

FIGURE 2.2: System F Signature

Syntax Signature

Figure 2.2 specifies the signature of the System F definitional interpreter implementation in Ocaml, translated from its mathematical notations described in Figure 2.1. The signature defines System F's expressions—comprising of abstractions and applications of variables (`var`) and type variables (`tvar`), and their respective values.

The implementation is *closure-based*, where a *closure* captures the environment, i.e., the evaluation context, and the abstraction that is evaluated. Therefore, `Closure` values are attain as a result of abstraction evaluations.

Finally, we include type `Int` as a primitive type for evaluations, along with binary operations `Binop`.

Evaluation Logic

The evaluation function is the encoding of the evaluation rules [16]. The `eval()` function has a signature of

```
environment -> exp -> value
```

where it takes an environment and an expression as its arguments, and returns a value. Some evaluations are straightforward: (1) integer and type expressions become their respective values, (2) variables and type variables are looked up from the environment (by T-VAR rule at Figure 2.3).

```
type environment =
      {types: (string * value) list;
      variables: (string * value) list}

let eval (env: environemnt) (t: exp) : value =
    match t with
    | Int i -> IntV i
    | Typ ty -> TypV ty
    | Var v -> lookup_var v env
    | ETVar tv -> lookup_ty tv env
    ...
```

As mentioned above, *abstractions* and *type abstractions* are evaluated to closures:

```
    ...
    | ETAbs (tv, exp)-> Closure (env, None, TVar tv, exp)
```

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-VAR)}$$

$$\frac{\Gamma, X \vdash t_2 : T_2}{\Gamma \vdash \Lambda X.t_2 : \forall X.T_2} \quad \text{(T-TABS)}$$

FIGURE 2.3: Some typing rules `t → t'` for System F

```
| Abs (v, ty, exp) ->
   Closure (env, Some v, propTy env ty lookup,
   propValTy env exp)
...
```

Evaluations of *applications* and *type applications* encapsulate the evaluation rules (Figure 2.4). Considering `ETApp`, or type applications, we evaluate the two expressions and pattern match them to a closure on the left-hand side and a type value on the right-hand side. Once the abstracted type variable `TVar "x"` is bound to a type value in the environment, `eval()` is called recursively on the body of the function with the updated environment (by `E-BETA2` at Figure 2.5). Variable applications are evaluated in a similar manner.

A Type Checker for System F

The type checker is used to find bugs and verify type soundness in programs [2]. Since the type checker is one of the static code analyses tools, it is important to understand its design before exploring how to test it.

Similar to the evaluator, the type-checker follows the typing rules specified in *Types and Programming Languages* [16]. The function `type_of_exp()` takes an environment and an expression, performs type

```
...
| ETApp (e1, e2) ->
   (match eval env e1, eval env e2 with
    | (Closure (cenv, _, TVar x, body), TypV ty') ->
       let new_env = {types = (x, TypV ty') :: cenv.types;
                      variables = cenv.variables}
       in
       eval new_env body
    | _ -> failwith "App to a non closure/tried to
    apply a non type"
   )
| App (e1, e2) ->
   (match eval env e1, eval env e2 with
    | Closure (cenv, Some x, _, body), v ->
       let new_env = { types = cenv.types
                     ; variables = (x,v) :: cenv.variables}
       in
       eval new_env body
    | Closure (cenv, None, _, body), _ ->
       eval cenv (App (body, e2))
    (*application to a polymorphic type, just continue*)
    | _ -> failwith "App to a non function"
   )
 ...
```

FIGURE 2.4: System F Application Evaluations

checking and returns a type of the expression with an updated environment.

```
let rec type_of_exp (env: ty_environment) (e: exp)
    : ty * ty_environment =
```

This section highlights the implementation for type checking type abstractions `ETAbs`. Type abstractions introduce the `ForAll` type that is yet to be discussed. The rest of code for the following interpreter can be found at this GitHub [7].

Typechecking for `ETAbs`, following the `T-ABS` rule [16] (Figure 2.3), is implemented as follows:

```
...
| ETAbs (tv, e) -> TForAll (tv, type_of_exp env e |> fst), env
...
```

$$(\lambda \mathtt{X}.\mathtt{t12})[\mathtt{T}_2] \rightarrow \{\mathtt{X} \mapsto \mathtt{T}_2\}\mathtt{t12} \qquad \text{(E-BETA2)}$$

$$\frac{\mathtt{t}_1 \rightarrow \mathtt{t}'_1}{\mathtt{t}_1[\mathtt{T}_2] \rightarrow \mathtt{t}'_1[\mathtt{T}_2]} \qquad \text{(E-TAPP)}$$

FIGURE 2.5: Some evaluation rules `t → t'` for System F

Since type variable `tv` is abstracted, the type of the abstraction expression can be described as "for all polymorphic type `tv`, this is the type of the function body".

Along with inferring types of expressions, the type checker follows the typing rules to ensure type soundness. For example, the following error is raised when a value of a wrong type is applied to a function parameter:

```
...
| App (e1, e2) ->
    ...
     | _ ->
        Monad.return_error ("Wrong application types.
        Failed the T-App rule with Type "
          ^ Utils.string_of_ty (fst e1_ty)) env
  in
...
```

Type checking is performed before function evaluations. This ensures that the expression follows all the typing rules, before the interpreter even attempts at evaluating it. As Robin Milner said, "Well typed programs do not go wrong" [12].

In summary, we have seen what System F is and how it can be implemented in OCaml and analysed via a type-checker.

Chapter 3

Testing with Interpreters

In this chapter, we describe our approach to revising the System F definitional interpreter to test static analyses.

3.1 Monads and Effects

The simple definitional System F interpreter from Section 2.2 returns only the result of executions. However, in order to test static analyses, we embed a program semantics collecting effect. To avoid introducing new bugs into the interpreter's evaluations, the embedding process must not adapt the evaluator's core logic. The mechanism for that are *monads*.

3.1.1 Background on Monads

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

FIGURE 3.1: Monad Type Signature in OCaml

```
module MaybeMonad = struct
  type 'a monad  = 'a option
  let return (x: 'a) : 'a monad = Some x
  let (>>=) (m: 'a monad) (f: 'a -> 'b monad) : 'b monad =
    match m with
    | None -> None
    | Some x -> f x
end
```

FIGURE 3.2: Maybe Monad in OCaml

Monads are generally used to wrap effectful computations, while requiring explicit lifting and binding of operations [3]. As a result, they can be used to *redefine* programming languages' semantics. Monads originally became popular through their use in Haskell, a purely functional programming language [3]. Since even printing is considered a side effect, monads were adopted to structure Haskell programs [3].

Traditionally, monads type signature specifies the use of two polymorphic functions called `return` and `bind`, as shown in Figure 3.1 [3, 10]. For example, the `Maybe` monad can act as an alternative to handling exceptions (Figure 3.2) - instead of raising errors, our monadic function would propagate `None`. Otherwise, we would get a result in the form `Some result`. In Figure 3.2, the return function returns successful output, whereas the bind function propagates the result or `None` if the function meets an error.

3.1.2 Monadic Interpreter for System F

This section details the revision of the simple System F definitional interpreter from Section 2.2. As monads can be used for structuring effects from computations, the aforementioned interpreter can be revised

to have the semantic collecting effect. We revise our interpreter to return the trace of function calls and respective environments of said calls, along with the result of the evaluation. The benefit of this approach is that it requires next to no adaptions to the interpreter's evaluation logic.

Signature of the Monad

Breaking down the signature of our monad below

```
type 'a t = 'a option * (log * environment) list
```

we have the `'a option` that acts similarly to the maybe monad from Figure 3.2. The signature for the function call and environment trace `(log * environment) list` will grow along with the evaluation steps of the program. The type `log` stands for the abstract type that stores the function calls and their respective arguments.

```
type log =
  | Eval of exp
  | ErrorLog of string
  | TypeOfExp of exp
   ...
```

As per tradition, our monad includes a return and bind operators [3, 10]. The return function lifts a pure result into a monad, where the `name` argument is of the appropriate `log` type. The bind function unwraps the result from the monad, evaluates the result, and re-wraps the new result back into an updated monad. The updated monad, in this case, appends the new semantic trace to the current, growing semantic trace.

```
let return (x: 'a) name state : 'a monad =
  (Some x, [(name, state)])

let (>>=) (m: 'a monad)
    (f: 'a -> 'b monad) : 'b monad =
  match m with
  | (None, semantics) -> None, semantics
```

```
match t with
(*Integer Evaluation *)
| Int i -> return (IntV i) name env
...
(* Type Applications*)
| ETApp (e1, e2) ->
   eval env e1 >>= fun lhs ->
   eval env e2 >>= fun rhs ->
   (match lhs, rhs with
    | (Closure (cenv, _, TVar x, body), TypV ty') ->
       let new_env = {types = (x, TypV ty') :: cenv.types;
                      variables = cenv.variables}
       in
       eval new_env body
    | _ -> Monad.return_error "App to a non closure/tried
    to apply a non type" env
   )
 ...
```

FIGURE 3.3: Monadic Refactoring of Evaluations

```
| (Some e, semantics) ->
  let res, new_semantics = f e in
  res, new_semantics @ semantics
```

Monadic Evaluation

Monadic parameterisation of the `eval()` function changes its signature to have return a type `value monad` rather than just `value`. The signature is updated by including the `return()` function in returning procedures, and passing the `name` variable (of type `log`) to record the expression we are evaluating. And finally, we replace "`let ... in ...`" statements with the bind operator to grow our monad (Figure 3.3). Otherwise, the code from non-monadic evaluations (Figure 2.4) and monadic evaluations (Figure 3.3) are almost identical.

Instead of raising exceptions and interrupting the semantics accumulation, `eval()` invokes `return_error`, defined below, instead.

```
let return_error s state: 'a monad =
    None, [(ErrorLog s, state)]
```

Since the logic of the monad handling is contained in the monad module, the refactoring does not interfere with the results or the evaluation logic.

Examples

To reiterate, the following monad collects a footprint of `eval()` function calls. The resulting data keeps track of what expressions are evaluated in what environments. In later chapters, this helps us test the type-checker via checking what types and variables were passed to expressions.

Consider the polymorphic identity function

```
let id_func = ETAbs ("X", Abs ("x", TVar "X", Var "x"))
```

Instantiating the polymorphic type `X` to integer, and applying the function to integer 1

```
eval empty_env (App (ETApp (id_func, Typ TInt), Int 1))
```

returns the result `Some (IntV 1)` and the following sequence of function calls

```
[Eval (ETAbs "X", Abs ("x", TVar "X", Var "x"))),
{types: []; variables: []};
Eval (Typ TInt),
{types: []; variables: []};
Eval (Abs ("x", TVar "X", Var "x")),
{types: ["X" , TypV TInt ]; variables: []};
Eval (Int 1),
{types: []; variables: []};
Eval (Var x ),
{types: ["X" , TypV TInt ]; variables: ["x" , IntV 1]]
```

3.2 Continuation Passing Style

Section 3.1.2 built a definitional interpreter with an embedded trace-collecting monad. However, this approach introduces a subtle problem

caused by the bind operator `(>>=)` making function calls before recording the trace, thereby introducing issues with call stack overflows. To make the bind operator tail-recursive, we utilise a different notion of monads - the *continuation passing style* (CPS) monad.

The following section, firstly, explains what is CPS and its virtues, namely solving issues with stack overflow. Finally, we refactor the interpreter to use CPS without amending our semantics collecting procedures.

3.2.1 Theory and Example

Functional programming languages, such as OCaml, provide means for continuations to be encoded as closures or first class anonymous functions [22]. Using this property, programs can be re-implemented with the Continuation-Passing Style (CPS) [22]. The overarching benefit of this approach is that all function calls become tail-calls, which avoids problems with overwhelming call stack growth [5].

In a CPS program, all functions take a continuation as extra argument. The continuation acts as the functional accumulator that stores the "rest of the computations" [5]. Thus, the return procedure of the function, once the function call arrives at a result, involves applying the continuation onto the said result.

For example, consider the Fibonacci function written using CPS at Figure 3.4. When calling the function, the anonymous identity function `(fun x -> x)` can be passed as the initial continuation. The type of the CPS Fibonacci function becomes `(int -> (int -> 'a) -> 'a)` making the function tail-recursive.

```
let rec fib n k =
   if n < 2 then k 1
   else fib (n - 1) (fun x ->
         fib (n - 2) (fun y ->
           k (x + y)))
```

FIGURE 3.4: CPS Fibonacci Function

3.2.2 Monadic Interpreter for System F in CPS

Continuation Monad

```
type ('a, 'b) monad = (('a, String.t) result -> 'b) -> 'b
```

FIGURE 3.5: CPS Monad Signature

Figure 3.5 portrays the signature of the CPS monad, where the type `result` is a built in OCaml type which returns `Ok of` some polymorphic type `'a` and `Error of` some polymorphic type `'b`. In other words, it acts as our Maybe monad, first introduced in Figure 3.2, implemented with type `result` instead. The monad is structured as a type of a CPS program itself, where `(('a, String.t) result -> 'b)` is the continuation.

As per tradition, the CPS monad also contains the return and bind operators [3, 10]. Then, the `return()` function, shown in Figure 3.6, applies the continuation `k` to some `Ok x` result, abiding by the general CPS return procedures (discussed in Section 3.2.1). The bind operator unwraps the result from its monad for evaluations, then re-wraps the updated result in some updated monad. The "unwrapping" in this case is done by defining a new continuation that would either update the contained result, or propagate a failure, as seen in Figure 3.6.

So far, the CPS Monad does not, yet, include the property of semantics

```
let return x k = k (Ok x)

let (>>=) x f k =
  let k' r =
    match r with
    | Ok z -> (f z) k
    | Error _ as x' -> k x' in
  x k'
```

FIGURE 3.6: Return and Bind Operators of CPS Monad

collection. Fortunately, the current implementation of the monadic interpreter allows for adding computational effects *modularly*, without having to alter the return and bind operators. Re-considering the CPS monad type, the intermediary value that accumulates the semantic trace of type `log list` can be integrated into type `'b` to be `(log list -> 'b)` instead.

The method for achieving dynamic collection can be found in Figure 3.7. The intermediary expressions are wrapped into a closure *thunk* that is run when given a unit type. Since all of the intermediary expressions have CPS monadic results (more specifically of type `(('a, String.t) result log list -> 'b) -> log list -> 'b`), the function will only be called with the first two arguments, namely the closure `thunk` and semantics trace `log`.

Note that the following method introduces *laziness* into our computations, as evaluating `(let y = f h z in g y)` in the monad does not reduce the expression, but rather allocates a closure with a reference to a closure that is result of `f h z` [23]. When recursively called, the construction induces nested non-tail calls, thus quickly growing the call stack [23]. However, laziness is handled by the CPS monad itself, as all function calls become tail calls.

```
let grow_collection thunk log k current_log =
      thunk () k (current_log @ log)
```

FIGURE 3.7: CPS Semantics Collection

```
let init_log r log' =
   match r with
   | Ok z -> Ok (z, log')
   | Error msg -> Error (msg, log')
```

FIGURE 3.8: Initial Continuation

The revision of the interpreter itself is minimal, only requiring wrapping return statements, or recursive calls, into *thunks*, which are then passed to the `grow_collection()` function (Figure 3.7).

Applications and Results

To run the CPS `eval()` function on some expression, we need an initial continuation to execute it with. We define a function called `init_log` as the initial continuation (Figure 3.8) with the following type:

```
(('a, String.t) result -> log list -> ('c, String.t) result).
```

Finally, we implement an evaluation function that does not require a continuation as an argument, and returns the result and footprint of our execution (Figure 3.9).

The aim of implementing a CPS monadic interpreter is to attempt achieving the same results found in Section 3.1.2, while working around

```
let eval_without_cps (t:exp) (env:environment) =
   let eval_res = eval_without_tc t env init_log [] in
   match eval_res with
      | Ok (z, log) -> Ok z, log
      | Error (msg, log) -> Error msg, log
```

FIGURE 3.9: Evaluation Without Continuation Passing

the CPS monadic infrastructure. Thankfully, the following results of evaluating applications of the System F identity function mimic the results found in Chapter 3.1.2.

```
eval_without_cps (App (ETApp (id_func, Typ TInt), Int 1)) empty_env

>  (Ok (IntV 1, {types = [("X", TypV TInt)];
variables = [("x", IntV 1)]}),
[Eval (ETAbs ("X", Abs ("x", TVar "X", Var "x")));
Eval (Typ TInt);
Eval (Abs ("x", TVar "X", Var "x"));
Eval (Int 1);
Eval (Var "x")])
```

3.3 Summary of the Techniques

In Chapters 2 and 3, we have elaborated on what System F, monads, and CPS are, as well as practically explored and experienced their virtues. After building a simple definitional interpreter for System F, we achieved a deeper understanding of its expressiveness, despite its concise syntax. Using monads, we were able to accumulate some semantics, specifically with regard to what environments are the expression evaluated at. This later proves useful when trying to test a type-checker of an industrial interpreter. One caveat was found: the possibility of overflowing the stack due to the monadic function calls being non tail-recursive. We solve this issue by refactoring our monad according to the CPS style and showing how both the result and the collected trace stay the same.

The code for the three interpreters from Chapters 2 and 3 is available on GitHub [7].

Chapter 4

Enhancing a Real-World Definitional Interpreter

Having explored the tools from the previous two chapters, we aim to incorporate those ideas in a real-world case study.

The following chapter discusses Scilla's definitional interpreter, highlighting its use of monads. Then, we describe the process of embedding of the collecting semantics monad within Scilla's interpreter. Finally, we introduce the use of strings as a universal format for storing the accumulated semantics from evaluated programs.

4.1 Scilla

Scilla is a new programming language for safe smart contracts. The language offers strong safety guarantees by means of type soundness of System F as foundational calculus [23]. In this section, we focus on Scilla's interpreter semantic design to properly navigate our future alterations.

```
let checkwrap_op op_thunk cost emsg k remaining_gas =
  if Uint64.compare remaining_gas cost >= 0 then
    op_thunk () k (Uint64.sub remaining_gas cost)
  else k (Error emsg) remaining_gas
```

FIGURE 4.1: Function for Monadic Gas Accounting

4.1.1 How Scilla Interpreter Uses Monads

The Scilla interpreter is constructed as a reference big-step monadic definitional interpreter [23]. In short, the result of the interpreter, of type `EvalRes Value`, is wrapped into the maybe monad. Therefore, `EvalRes` returns successful computations, while failure is propagated and made explicit.

Additionally, Scilla's monad acts as a state monad, allowing for one of the key computations: tracking resource consumption. Any deployment or interaction with a contract would require the emitter to pay (in virtual funds) a specific amount of *gas* [23]. If the user's allotted amount of gas does not cover the cost of the future execution, an *out-of-gas* failure is raised. Scilla's designers perform gas accounting monadically, thus doing it without altering the interpreter's logic.

Referring back to the CPS monad signature in Figure 3.5, integration of gas into the monad is done via transforming the type `'b` in Scilla's monad into `Gas -> 'b`. Thus, computations returning `EvalRes Value` now become functions expecting a certain amount of gas, and can only run once the the sufficient amount of gas is provided.

Figure 4.1 displays the code for monadic gas deduction. Before passing an `Error()` to the continuation, or evaluating the closure `op_thunk` with updated intermediary gas value, the function first checks whether

the remaining gas covers the cost of the execution. The drawback of allocating closures to intermediate expressions is introduction of *laziness* into evaluation.

4.1.2 CPS Evaluator

As discussed in Section 3.2.2, issues with laziness, which cause call stack overflow, can be solved by rewriting programs according to CPS. For this reason, the monad is reinterpreted as a CPS monad, making all function calls tail-calls [5]. As the revision involves only the monad, it does not affect the core interpreter's logic.

It is also worth mentioning that Scilla is not in full CPS, as that requires serialisation of closures [23] (Figure 4.2). As such, the interpreter contains components that "cut" the CPS execution (Figure 4.3). Figure 4.3 details a function that fully evaluates an expression with a fixed continuation `init_gas_kont`, the result of which is passed onto the callee's continuation `k`. In Chapter 5, we will demonstrate how constraints like these make monadic treatment for real-world projects difficult.

```
type t =
...
| Clo of (t ->( t, scilla_error list, Gas -> SemanCollect ->
              ( (t * (LType.TIdentifier.Name.t * t) list)
              * Gas * SemanCollect,
                scilla_error list * Gas * SemanCollect)
              result)
            CPSMonad.t)
...
```

FIGURE 4.2: Explicit Type Declaration of Closures

```
let exp_eval_wrapper_no_cps expr env k gas =
  let eval_res = exp_eval expr env init_gas_kont gas in
  let res, remaining_gas =
    match eval_res with
      |  Ok (z, g) -> (Ok z, g)
      | Error (m, g, l) -> (Error m, g)
  in
  k res remaining_gas
```

FIGURE 4.3: Expression Evaluation Function without CPS

4.2 Retrofitting Scilla Interpreter for Testing

Modularity of Gas Accounting

```
| GasExpr (g, e') ->
      let thunk () = exp_eval e' env in
      let%bind cost = fromR @@ eval_gas_charge env g in
      let emsg = sprintf "Ran out of gas.\n" in
      checkwrap_op thunk (Uint64.of_int cost) (mk_error1 emsg loc)
```

FIGURE 4.4: `Gas` expression Handling

To properly explain how we derive the methods to embed the collecting semantic monad, we explore the code behind the gas-aware monad. After doing so, we demonstrate how we seamlessly combined the two monads together.

Scilla's CPS monad does not explicitly declare `Gas` in its type as shown below.

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

The `Gas` type is later inferred due to the monad being used as a gas-aware monad, thus expanding some polymorphic type `'c` of the monad to `Gas -> 'c` instead. Scilla defines a gas-aware monad module which includes utility functions for gas handling throughout the steps of evaluation. The interpreter handles gas separately from other expression evaluations by

defining its own gas expression `GasExpr` (Figure 4.4). In summary, gas accounting is designed to be fully modular from the interpreter.

Modularly Embedding Semantic Collecting

In this section, we aim to design a generalised procedure of dynamically collecting the trace of every expression evaluated.

Let us denote `SemanCollect` to be the type of our data structure that stores accumulated semantics. First, we implement `SemanCollect` as its own intermediary value, completely separate of `Gas`. Thus, we aim for the final type of the monad to look like

```
(('a, 'b) result -> Gas -> SemanCollect -> 'c)
    -> Gas -> SemanCollect -> 'c
```

Additionally, we need a procedure that appends new traces onto `SemanCollect`. Similarly to how gas is handled (Figure 4.1), we define a function that, given a closure `thunk`, grows the intermediary `SemanCollect` structure with newly recorded data (Figure 4.5). Finally, we run the closure `thunk` with the new `SemanCollect`.

```
let update_monad_log thunk seman_collect k
    remaining_gas current_seman_collect =
        thunk () k remaining_gas
            (update_seman current_seman_collect seman_collect)
```

FIGURE 4.5: Accumulating Semantics

As discussed in Section 4.1.2, `init_gas_kont` is passed as an initial continuation in expression evaluation function calls that "cut" the CPS structure. To make `SemanCollect` explicit in continuations, we pass it as an argument in `init_gas_kont` (Figure 4.6). Once passed, `SemanCollect` is then inserted into the specialised type `result` of the closures in the program.

```
let init_gas_kont r gas' seman_collect' =
  match r with Ok z ->
  | Ok (z, gas', seman_collect')
  | Error msg -> Error (msg, gas', seman_collect')
```

FIGURE 4.6: Enhanced Initial Continuation

Finally, the functions is incorporated into the expression evaluations (`exp_eval`) by wrapping either a return procedures or a recursive calls into a closure `thunk`. `thunk` is then passed to a function `collecting_semantics()` which handles some filtering and updates to the `SemanCollect` variable. Once `SemanCollect` is updated, we run `thunk`, i.e. the respective return or recursive calls. The following code snippet shows an example `Let` expressions revisions, where utility functions `let_semantics` convert the expression into strings - our chosen universal format for storing trace semantics (discussed in the next section 4.2.1).

```
| Let (i, _, lhs, rhs) ->
  let%bind lval, _ = exp_eval lhs env in
  let env' = Env.bind env (get_id i) lval in
  let thunk () = exp_eval rhs env' in
  collecting_semantics thunk loc
          (let_semantics i lhs lval)
```

4.2.1 Strings as a Universal Format

When trying to collect the trace of all expressions passed, intuitively, we aim to collect and store the the exact data structure of expressions to avoid loss of information. Yet, we choose to store our data as strings instead. This section reasons why strings are the chosen as the universal format of storing data, describes their advantages, but also the restrictions.

Firstly, converting the results to strings using defined pretty printers make the output readable and therefore easier to understand. Then, we can write a parser to translate the strings back into their original data structures, avoiding any loss of information. Additionally, when parsing the strings, we can re-structure or select just the data we need for testing specific static analyses. As strings are a commonly used data type, there are not steep learning curves to overcome.

Furthermore, there are limitations in the Scilla interpreter due to which we cannot store expressions as they are. As mentioned in Section 4.1.2, closures in Scilla are not serialised, i.e., abstract type definitions are made explicit in OCaml. This exposes the type declarations of `SemanCollect` and `Gas` that exist in Scilla closures and abstractions (Figure 4.2). Defining abstract data structures in OCaml limits us to using only monomorphic types or polymorphic types declared in the module, i.e. we cannot define some abstract type `SemanCollect` as an `'a list`. While this poses no problem for gas, as it already resembles primitive types, `SemanCollect` has a limited range of types it can be.

Additionally, all expressions have a type defined in module `EvalSyntax` - a specific instance of the general `Syntax` module. `Syntax` module is then built upon the general `Literal` module and other modules combined. In short, the dependency of modules looks like `Literal` $\rightarrow$ `Syntax` $\rightarrow$ `EvalSyntax`. The designers for Scilla specifically deter recursive module definitions that involve `Eval` modules (such as `EvalSyntax`), only permitting `Eval` modules to depend on the base modules. Thus, it would not be possible to refer to a specific type definitions from `EvalSyntax` for defining `SemanCollect` in the `Literal` module.

In summary, Scilla's design prevents us from storing expressions as they are without restructuring the project's dependencies. As such, we store data as strings and reap the benefits of the data type's readability, simplicity, and familiarity.

Resulting SemanCollect

Our goal for this section is to be able to incorporate the semantics collecting monad to at least record every expression evaluated without interfering with evaluations. The following section bears the fruit of our implementations, as we show our first results of tracing a program's execution in the production scale interpreter.

Consider the example program, written in Scilla, at Figure 4.7.

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

FIGURE 4.7: Example Program in Scilla

When running the evaluator and extracting the footprint of expression evaluations, we get the result shown below. Mapping the resulting trace to the original program (Figure 4.7) allows us to see how having a monad collect information in between function calls provides with the an ordered and concrete trace of semantics.

```
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))
```

4.2.2 Specialising Collecting Abilities

Now that we have finally built a system to collect some data, we can redefine some collection processes to get further insight into the program. This section details how we can specialise our current implementation to allow for collection of the types of data flows into each variable.

To do so, we update `SemanCollect` from its previous form, of being a `(string list)`, that stores trace of semantics as strings. `SemanCollect` will now contain a dictionary of variables and data passed to them, as well as the pre-defined trace of expression evaluations. The type of `SemanCollect` is now:

```
((String.t * String.t) List.t * String.t List.t)
```

Since only `Let` expressions introduce new data flows into variables, the collection procedure is updated accordingly.

```
| Let (i, _, lhs, rhs) -> ...
  collecting_semantics thunk loc
    (new_flow (Var i) (fst lhs),
    (let_semantics i lhs lval))
```

The function `new_flow` updates the dictionary with new data from `lhs` to flow into `Var i`. Once the function evaluates and we have populated our dictionary with all variable definitions, we can experiment with re-structuring our data to get the information we need.

For example, after running the example program mentioned above (Figure 4.7), we can traverse the dictionary to find all variables and literals that flow into our variable definitions.

```
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))
```

Chapter 5

Case Study

Having developed the general framework for dynamic semantics collection, we explore strategies for populating and refining our `SemanCollect` for testing specific static code analyses. One of such static code analyses is a static type checker.

Scilla has its own type checker which statically checks that contracts are well-typed [23]. Ensuring type-soundness of contracts is crucial for providing safety guarantees in block-chain programming. Scilla designers argue that well-typed contracts do not go wrong but can fail with *expected* failures only [23]. As such, it is important to make sure that the type checker is, indeed, bug free.

In this section, we demonstrate how we integrated type flow collection into our monad, applying it to testing the Scilla's type checker results.

5.1 Accumulating Type Flows

Scilla programs come in several shapes: type annotated (after type checking) and unannotated (that are evaluated). Our goal is to make sure that

the former is correct with regard to running with the latter, which is done by checking if the concrete flow of literals (which are typed) is coherent with the ascribed types of named variables. In essence, we must check only expressions of type `T` flow into an expression `x` of type `T` (`x:T`).

5.1.1 Enhanced Monadic Collection Procedures

First, we define our procedure to extract types from expressions. Scilla programs that are evaluated are not type annotated. Therefore, we find out the types of our variables by extracting the types of literals that flow into them. For example, given an expression

```
let x = Int32 1 in
let y = x in
y
```

we can only know the type of `y` by knowing `(Int32 1)` flowed into it through `x`.

Referring back to `SemanCollect`, let's redefine its type to contain the types of the expressions stored (Figure 5.1). We include type `option` as many of our expressions are not typed when first collected.

```
(((String.t * LType.t option)
* (String.t * LType.t option)) List.t * String.t List.t)
```

FIGURE 5.1: Type of SemanCollect

With our new type for `SemanCollect`, we update our flow collection procedure `new_flow` (Figure 5.2). Originally, `new_flow` only collected a flow of an expression `v2` into `v1` by storing them as pairs `(v1, v2)`. We include a type parameter that records the type of `v1` (either `None` or `Some ty`). The type of `v2` is only known if it is a literal, i.e. a constant, the type of which we can easily extract.

```
let new_flow v1 v2 (ty: SType.t option):
((String.t * LType.t option) * (String.t * LType.t option)) =
    let v1' = (no_gas_to_string v1, ty) in
    match un_gas v2 with
    | Literal l ->
        begin
        match l with
        (*Closures don't have serialised types*)
        | Clo _ -> (v1', (no_gas_to_string v2, None))
        | _ ->
            match literal_type l with
            | Ok ty' -> (v1', (no_gas_to_string v2, Some ty'))
            | Error _ -> (v1', (no_gas_to_string v2, None))
        end
    ...
```

FIGURE 5.2: `new_flow` function

5.1.2 Propagation of Logic in Evaluator

With our new collecting procedures ready, we plan where to insert them throughout the evaluator to collect the flows. For this, we define more concretely what we mean by expressions flowing into other expressions:

Definition 1. An expression `e1` *flows* into another expression `e2` when we see the evaluator run `bind()` binding the evaluated expression `e2` onto `e1` in some environment. This includes what expressions flow into function parameters and its local variables during applications.

Abiding by definition 1, we propagate the use of `new_flow` in `Let`, `Fun`, `App`, and `MatchExpr` expressions, all of which bind variables in local and global environments.

We highlight how we incorporate the collecting procedures for applications `App` due to the complexity of handling closures. Since the Scilla interpreter is closure-based, its `Fun` expressions are evaluated to closure `Clo`. Thus, when running an application of some variable `x` to some function `f`, `f` is only stored as a closure in the global environment forbidding

```
(* Expected result: 1 *)
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
```

FIGURE 5.3: Example Program in Scilla 2

us to access the function's parameters and their types. In other words, `SemanCollect` cannot collect what function parameters is the expression being passed to.

To solve this issue, we store the `new_flow` function in the closure as well. When the function is called for evaluation, it runs the collecting procedure that records the function parameter and its type. To collect what expressions flow into the function parameters, we record the identifiers of the expressions in `App` expression evaluations before running the closures.

5.1.3 Results

Consider the following Scilla program that includes applications in Figure 5.3. Running the enhanced monadic evaluator on the program gives us the following set of flow accumulations:

```
(Variable x, ___),(Lit (Int32 42), Int32)
(Variable f, ___),(Fun (Var z: Int32), Int32)
(Variable a, ___),(Lit (Int32 1), Int32)
...
```

Note how only literals and function parameters have their types predefined. For the rest, we must infer them from the literals that have flown into them. Doing so, we get the following collection:

```
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))
```

5.2 Testing Scilla's Static Type Checker

Out strategy for testing Scilla's static type checker involves confirming the inferred type of expression `x` is correct via runtime. In essence, we can do that by checking for `x:T` - only expressions of type `T` float into it. From our results in section 5.1.3, this can be done by simply checking if the list of expressions flowing into a named variable match types.

Re-evaluating the program at Figure 5.3 gives us

```
...
Variable b: Int32 <- ((Variable x: Int32) <- (Lit (Int32 42): Int32))
    => Flows type check
Variable c: Int32 <- ((Variable d: Int32) <- (Lit (Int32 2): Int32))
    => Flows type check
```

Running a similar program to the one in Figure 5.3, but replacing `(let d = Int32 2)` with `(let d = "abc")`, we get the following type error:

```
...
Variable b: Int64 <- ((Variable x: Int64) <- (Lit (Int64 42): Int64))
    => Flows type check
Variable c: Int64 <- ((Variable d: String) <- (Lit (String "abc"): String))
    => Flow does not type check
```

The following framework is run on 105 Scilla programs provided by the Scilla designers as test programs. All of the 105 programs were considered "good", i.e., statically type sound. Our results concluded that type soundness held up dynamically for all of the programs as well. Therefore, we are not able to find any bugs in the type checker from running these programs.

In a larger context, testing the type checker with 105 human written programs may not yield substantial results. In order to comprehensively test the type checker, we need a random Scilla program generator that could generate a good distribution of semantically correct programs [13].

5.3 Running the program

The code for the progress implemented in this chapter exists in a public fork of the Scilla interpreter[1].

After following the compilation steps articulated in the `README` file, running the `eval-runner` binary with the flag `-output-seman`, followed by a path to a JSON file, will output the results of the dynamically collected data into the specified file.

Here's an example of a command of execution:

```
eval-runner -gaslimit 10000 -output-seman a.json -libdir
src/stdlib tests/eval/good/let.scilexp
```

[1] https://github.com/tramhnt99/scilla

Chapter 6

Discussion

6.1 Future Work

The focus of this project is to provide a proof of concept of refactoring a production scale definitional interpreter with a semantics collecting monad to test static code analyses. We successfully extracted some dynamically collected semantics which we used to confirm the results of Scilla's static type-checker. The future work can focus on implementing a parser that would allow us to broaden the scope of the kind of static analyses we can test.

Our immediate next task is to implement a parser that parses the resulting strings, storing dynamically collected semantics, back to their respective structures without losing any precision. This prevents us from being limited by handling collected data as strings, and provides us with broader possibilities of the kinds of analyses we can test.

An example of such analysis is the type-flow analysis used for monomorphisation [15]. *Monomorphisation* is a process by which function types are made precise because polymorphic functions are difficult to reason and slow to compile. A monomorphisation pass uses type-flow analysis,

which statically determines what types certain variables might take on during run time. As such, testing type-flow analysis is a direct extension from our current strategies of testing type checkers.

Additionally, the following framework can be further developed to test a newly developed static program analysis tool called *CoSplit* [17]. CoSplit statically infers properties of smart contracts that are used for maximising parallelism to achieve scalability in blockchain protocols [17]. As CoSplit runs static analysis on contracts written in Scilla, we can extend this project's program for future testing of CoSplit.

6.2 Conclusion

In this project, we presented a detailed approach of incorporating a semantics collecting monad into the real world definitional interpreter. We started by demonstrating monadic parameterisation of a simple System F interpreter that does not have a monad. By embedding the CPS monad into our new monadic System F interpreter, we presented how we can get the same trace of semantics from the tail-recursive monad. Finally, we integrated the ideas to enhance the Scilla interpreter for testing its static analyses.

Integrating our monad into Scilla required overcoming many challenges. Firstly, we devised a suitable notion of our monadic semantics to integrate with Scilla's non-serialised closures and abstraction definitions. Those definitions required us to make explicit the type of the data structures storing our semantics, which influenced our choice of storing them as strings. Putting our framework together, we achieved a proof of concept for recording dynamic semantics of programs in Scilla.

The results of chapter 5 which show how our concrete semantics checked the static type checker, highlight our achievement of successfully monadically refactoring a production-scale interpreter to extract the necessary dynamic analysis for testing static analyses. Overall, we hope the project's novel approach for testing static analyses sows the seeds for new opportunities within the field programming languages.

Bibliography

[1] Peter Aldous and Matthew Might. “Static Analysis of Non-interference in Expressive Low-Level Languages”. In: *Static Analysis - 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings*. Ed. by Sandrine Blazy and Thomas P. Jensen. Vol. 9291. Lecture Notes in Computer Science. Springer, 2015, pp. 1–17 (cit. on p. 1).

[2] Nada Amin and Tiark Rompf. “Type Soundness Proofs with Definitional Interpreters”. In: *SIGPLAN Not.* 52.1 (Jan. 2017), 666–679. ISSN: 0362-1340 (cit. on p. 10).

[3] Nick Benton, John Hughes, and Eugenio Moggi. “Monads and Effects”. In: *International Summer School on Applied Semantics* (2000), pp. 42–122 (cit. on pp. 14, 15, 19).

[4] Patrick Cousot and Radhia Cousot. “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints”. In: *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 1977, pp. 238–252 (cit. on p. 1).

[5] Olivier Danvy. “Back to direct style”. In: *Science of Computer Programming* 22.3 (1994), pp. 183–195 (cit. on pp. 3, 18, 25).

[6] Jean-Yves Girard. "Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur". PhD thesis. Éditeur inconnu, 1972 (cit. on p. 5).

[7] Tram Hoang. *Enhanced System F Interpreters*. https://github.com/tramhnt99/Monadic_SystemF_Compiller. 2021 (cit. on pp. 11, 22).

[8] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. "Partial Dead Code Elimination". In: 29.6 (June 1994), 147–158. ISSN: 0362-1340 (cit. on p. 1).

[9] 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 p. 3).

[10] Jan Midtgaard and Anders Móller. "QuickChecking Static Analysis Properties". In: *Software Testing, Verification and Reliability* 27.6 (2017). e1640 (cit. on pp. 1, 2, 14, 15, 19).

[11] 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 p. 2).

[12] Robin Milner. "A Theory of Type Polymorphism in Programming". In: *J. Comput. Syst. Sci.* 17.3 (1978), pp. 348–375 (cit. on p. 12).

[13] 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*. AST '11. Waikiki, Honolulu, HI, USA: Association for Computing Machinery, 2011, 91–97 (cit. on p. 37).

[14] M. H. Pałka, K. Claessen, A. Russo, and J. 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 pp. 6, 7).

[15] Gabriel Phoenix Petrov. "Type and Control Flow Analysis of Functional Programs". Bachelor's Thesis. 2021 (cit. on p. 38).

[16] Benjamin C. Pierce and C. Benjamin. *Types and Programming Languages*. MIT press, 2002 (cit. on pp. 1, 3, 5–7, 9–11).

[17] George Pîrlea, 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 (PLDI '21)*. To appear. ACM, 2021 (cit. on p. 39).

[18] John C. Reynolds. "Definitional Interpreters for Higher-Order Programming Languages". In: *High. Order Symb. Comput.* 11.4 (1998), pp. 363–397 (cit. on p. 2).

[19] John C Reynolds. "Towards a theory of type structure". In: *Programming Symposium*. Springer. 1974, pp. 408–425 (cit. on p. 5).

[20] David A. Schmidt. "Trace-Based Abstract Interpretation of Operational Semantics". In: *Lisp and symbolic computation* 10.3 (1998), pp. 237–271 (cit. on p. 3).

[21] 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 p. 3).

[22] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. "Scilla: a smart contract intermediate-level language". In: *arXiv preprint arXiv:1801.00687* (2018) (cit. on p. 18).

[23] 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. 4, 5, 20, 23–25, 32).

[24] Olin Shivers. "Control-flow analysis of higher-order languages". PhD thesis. PhD thesis, Carnegie Mellon University, 1991 (cit. on p. 1).