giaco.ml

an interpreted EDSL written in OCAML

Mario A. Barbara
Andrea Cancellieri

Part I

Introduction

giaco.ml is an imperative (but also functional) EDSL interpreter written in OCAML.

The interpreter is capable of handling expressions, commands and declarations all within a program.

A program can be parsed into valid code from a file/string thanks to reflection.

Static taint analysis can be performed, guards have been also added to the Reflect command as a form of protection, similar to perl.

The main execution functions are: `interpret` (or `interpret'` which returns also the declarations' output), `reflect` and `taint_analysis` which all operate on a `Prog(ds,cs)` construct.

Part II

Design Choices

The interpreter works with 3 domains:

generic this domain is shared amongst the syntax (external) and semantic (internal) domains

syntax this domain contains all the domains accessible to the user. it features 3 sub-domains

- **expressions** these are entities which may be mapped directly to a value
- **commands** these are entities which allow modifying the global state of the program, though allocating new memory is not allowed
- **declarations** these are entities which allow allocation of new memory in the global state

semantic this domain is used for representing the internal state of the interpreter.

- **evaluatable values** these are values that may be directly evaluated from an expressions. they represent the base internal type for the interpreter. Basic allowed types are Integer, Boolean, Float, String and Subprograms.
- **environment** this is a mapping representing internal non-mutable state. It maps variables (identifiers) to values.
- **store** this is a mapping representing internal mutable state. It allows aliasing of 2 identifiers for one value. It maps addresses (pointers or locations) to values.
- **denotable values** these are values contained in the environment. Basic allowed types are Integer, Boolean, Float, String and Locations.
- **mutable memory values** these are values contained in the store. Basic allowed types are Integer, Boolean, Float, String.

The interpreter itself is implemented by 3 main functions:

eval this function basically maps expressions to evaluatable values

cval this function maps commands to a changed internal state (store)

dval this function maps declarations to a new internal state (environment + store)

Some important choices:

- memory is viewed as a mapping. it is therefore impossible to modify in ocaml. changing the type signature of the interpreter would allow this
- because memory is a mapping, new locations are simulated by randomly selecting a new location based on given store size
- regular expressions are used in reflection, so the Str module must be available
- fixed points are calculated through ocamls's `function` and `let rec` operations
- to ease debugging, the interpreted is not compiled but interpreted in an ocaml shell with the `#use "file.ml"` construct
- since the program is interpreted, `let and` between multiple files is not available, this has been fixed by using mutable state pointers to future functions. These pointers will be updated when the function is created.
- many utility functions have been created to ease debugging. most notable are `emptyenv emptystore tenv' tstore' tnew'` which simplify dealing with memory

Part III

Usage

the interpreter can be loaded up with

```
> #use "giaco.ml";;
```

and subsequent testing may be performed:

```
> let e = Plus(Int(3), Int(4)) in eval e emptyenv emptystore
- : evalue = EInt 7
```

Chapter 1

EXPRESSIONS

EXAMPLE	DESCRIPTION
Int(3)	basic integer
Str("hello world")	basic ASCII string
Bool(true)	basic boolean
Float(4.5)	basic float
Lambda("x", <exp containing x>)	typical function
RecLambda("f", "x", <exp containing f and x>)	typical recursive function
Rec("f", Lambda(....))	just another way to define recursive lambdas
Proc(["x";"y";"z";...], Block(...))	this is a procedure, check the commands section
IfThenElse(Bool(true), .., ..)	control flow element
Var("x")	this is a way to retrieve an immutable variable's content
LetIn("x", e1, e2)	this is a way to nest functional blocks and scopes
Val("x")	this is a way to retrieve a mutable variable's content
Plus(e1, e2)	plus function, applies to: Int, Str, Float
Multiply(e1, e2)	multiply function, appliest to: Int, Str, Float
Apply(e1, e2)	typical function application, e1 is of type: Lambda, RecLambda, Rec
Equals(e1, e2)	like C's `==`
Greater(e1, e2)	like C's `>`
Not(e)	like C's `!`
Or(e1, e2)	like C's `||`
And(e1, e2)	like C's `&&`
Len(Str(...))	gets the length of a St
Sub(Str(...), i, j)	gets a substring. i and j of type Int.
Lower(Str(..))	reduces a string to lowercase, like Python's `lower()`
Upper(Str(...))	reduces a string to uppercase, like Python's `upper()`
Trim(Str(...))	trims whitespace from a string, like Python's `s.trim()`
Replace(<string to be replace>,<replacer string>,<string>)	replaces a string with another string in a string, like Python's `s.replace()`

Chapter 2

COMMANDS

EXAMPLE	DESCRIPTION
Assign("x", e)	this changes the mutable value for the variable "x". e is an expression
Block(d, c)	this is an imperative block with nested scope. d is a declaration, see its section for more detail
Call(p, [e1;e2;e3;..])	this is an application of an imperative procedure. p is of type Proc (check the expressions section)
While(e, c)	like C's `while(e){c}`, e is an expression and c a command
CIfThen(e, c)	like C's `if(e){c}`
CIfThenElse(e, c1, c2)	like C's `if(e){c1}else{c2}`
CSeq(c1, c2)	like C's `;` it allows concatenation of commands
CSkip	like C's `void` and Python's `pass`, it does nothing
Reflect(Str(...))	reflection, see the reflection section

Chapter 3

DECLARATIONS

EXAMPLE	DESCRIPTION
New("x", e)	this allocates a new mutable variable of value e (an expression)
DSeq(d1, d2)	allows concatenation of declarations
DSkip	does nothing

Part IV

String extension

A few functions have been added to deal with the domain of strings. Functions such as these are taken from the Python langauge, which has a **very** extensive and popularstandard library.

- length comparison (`Greater`)
- concatenation (`Plus` has been extended to allow this)
- substring (`Sub`)
- repetition (`Multiply` has been extended to allow this
- length (`Len`)
- lowercase (`Lower`)
- uppercase (`Upper`)
- trim (`Trim`), trims all whitespace
- Replace (`Replace`)

Check the examples section for some examples

Part V

Reflection extension

Reflection consists of allowing any string to be evaluated by the interpreter on the fly. In Python this is akeen to the `eval` function. This is also the most essential step to having a good interpreter: the interactive console for Python, one of the most popular interpreted languages, is often called **R.E.P.L.** (Read Eval Print Loop).

The syntax of *giaco.ml* has been extended with the `Reflect` command, which allows on the fly "evaluation" of commands. Unfortuately our language's command syntax is recursive, and furthermore the `CIfThenElse` command uses expressions as boolean conditions, which are also recursive. Therefore, a full blown parser needed to be built to give a string some depth (such as that of an AST).

The reasoning is as follows:

1. a function called `next_unit` is charged with grabbing the first word up until a `(` or `)` or `,` or multiple consecutive repetitions.

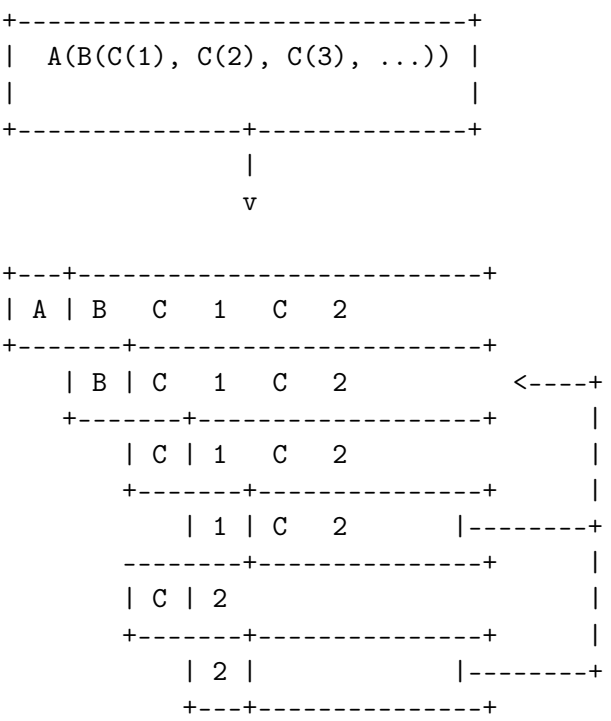
2. to get the command to match against, `next_unit` is called on the string and the result is matched against some constants, taken from the language's syntax

3. to get a command's arguments (which may be recursive and contain any amount of `( ) ,`, caution must be taken to correctly identify the argument boundaries, which are all separated by a `,` comma. 2 options are given:

   iterative by counting the number of open parentheses matched thus far, and decreasing each time a closed parentheses is found, it is possible to correctly identify the recursive structure of the syntax.

   recursive (but faster) since we know the amount of parameter each command needs, it is simply required to recursively reflect upon the arguments' string as many times as needed. Of course, each time a command is consumed, it shall return the arguments' string, so as to allow its father to continue looking for arguments.

Our interpreter implements the recursive and faster technique. Here is a simple ditaa drawing to illustrate the flow of this technique:



Part VI

Taint-Analysis extension

Static taint analysis consists of understanding how much damage some unsafe elements (of undefined value but defined nature) will yield. A classic example is an unsanitized input on a HTML form, which may result in an SQL Injection attack and damage your company's most valuable assets.

In our simple language, we have no operations that deal with the outside world (yet). We are thereforce forced to ask the user to label some variables in the environment and store as `Clean` or `Dirty`. Afterwards, we will analyze a program and check the `Taint` for every possible variable assignment. The semantic domains have been revisited, allowing memory (environment and store) to only contain tainted values (or store locations, in the environment's case).

Our analysis is based on 2 simple concepts:

pure evaluation a `tor` function will take 2 taints and return `Dirty` if one of them is as well, otherwise `Clean`. This process can be lazy.

- All constants are `Clean`
- If a function is involved (such as a `Lambda`) then the formal parameters are identified as `Clean` (as they cannot be expressions) and then the body is analyzed. If the body is clean, the function is clean
- A function application requires a `tor` amongst the analysis of the function itself and the passed parameter.
- If a condition is involved, then 2 outputs are possible. If the condition is `Dirty`, that means the attacker may choose either output and (regardless of the output's default taint) will result in a `Dirty` value. If the condition is `Clean`, then either output may occur during execution, so they must be passed to `tor`.

imperative state change all possible assignments in a command are gathered. Only the latest possible assignments matter (if i set `x` to `Dirty` and then `Clean` it is `Clean`).

- Afterwards, we check whether 2 branches are possible: if they are, a `tor` function must be applied to all assignments of same key, merging the 2 branches.
- If the branches are subject to a condition (such as in a `CIfThenElse`) then a `Dirty` condition will mean an attacker may choose amongst any of the 2 branches, therefore dirtying all assignments of shared key (amongst the 2 branches). If the condition is `Clean`, then the normal merge has already evaluated taint with `tor`.

Part VII

Examples

check test.ml for some code examples.

Chapter 4

Numbers

INPUT	OUTPUT
Int(5)	EInt 5
Float(133.7)	EFloat 133.7
Plus(Int(1), Int(2))	EInt 3
Multiply(Float(2.5),Float(10.0))	EFloat 25
Greater(Int(3),Int(5))	EBool false

Chapter 5

Booleans

INPUT	OUTPUT
Bool(true)	EBool true
Not(Bool(true))	EBool false
And(Equals(Float(4.5),Float(4.6)),Equals(Float(0.1),Float(0.1)))	EBool false
Or(Equals(Float(4.5),Float(4.6)),Equals(Float(0.1),Float(0.1)))	EBool true

Chapter 6

Strings

INPUT	OUTPUT
Str("hello world")	EStr "hello world"
Plus(Str("hello "),Str("world!"))	EStr "hello world!"
Multiply(Str("abc"),Int(10))	EStr "abcabcabcabcabcabcabcabcabc"
Len(Multiply(Str("abc"),Int(10)))	EInt 30
Greater(Str("two"),Str("three"))	EBool false
Sub(Str("threeeeeeee"),Int(2),Int(10))	EStr "reeeeeeee"
Upper(Str("im so lonely"))	EStr "IM SO LONELY"
Lower(Upper(Str("im so lonely")))	EStr "im so lonely"
Trim(Str(" italia "))	EStr "italia"
Replace(Str("hello"),Str("goodbye"),Str("hello world!"))	EStr "goodbye world!"

Chapter 7

Functional Control Flow

INPUT	OUTPUT
IfThenElse(Bool(true), Int(1337), Str("i am"))	EInt 1337
IfThenElse(Not(Greater(Str("bob"),Str("mouse"))),Str("ciao mondo"),Int(5))	EStr "ciao mondo"

Chapter 8

Functional Blocks

INPUT	OUTPUT
Var("x")	EInt 20
xxx = LetIn("a",Int(3),Multiply(Var("a"),Var("a")))	EInt 9
LetIn("a",Int(5),(LetIn("b",xxx,LetIn("c",Int(6),Plus(Var("a"),Plus(Var("b"),Var("c"))))))))	EInt 20

Chapter 9

Functional Subprograms

INPUT

Apply(Lambda("x", Plus(Var("x"), Int(1))), Int(99))

Apply(RecLambda("sum", "x", IfThenElse(Equals(Var("x"), Int(0)), Int(1), Multiply(Var("x"), Apply(Var("sum"), Plu

Chapter 10

Imperative State Change

INPUT	VARIABLE OUTPUT
Val("y")	EInt 10
Assign("y", Plus(Val("y"), Val("y")))	EInt 20

Chapter 11

Imperative Control Flow

INPUT

Val("y"), Val("z")

CIfThenElse(Not(Equals(Val("y"),Int(11))), Assign("y", Int(50)))

While(Not(Equals(Val("y"), Int(100))), CSeq(Assign("y", Plus(Val("y"), Int(1))), Assign("z", Plus(Val("z"), Int(1)))))

Chapter 12

Imperative Blocks

INPUT	OUTPUT
Val("y"), Val("z")	EInt 10, EInt 0
Block(New("z", Int(1000)), Assign("y", Plus(Val "y", Val "z")))	EInt 1010, EInt 0

Chapter 13

Imperative Subprograms

INPUT	OUTPUT
Val("y"), Val("z")	EInt 10, EInt 0
f = Proc(["z"], Block(DSkip, Assign("y", Val("z")))))) in Call (Val "f", [Val "z"])	EInt 0, EInt 0

Chapter 14

Declarations

INPUT	OUTPUT
Val("y"), Val("z")	Failure 'y' not in environment, Failure 'z' not in environment
DSeq(New("y", Int(10)), New("z", Int(0)))	EInt 10, EInt 0

Chapter 15

Reflection

INPUT	OUTPUT
Val("y")	EInt 10
ereflect ("Plus(Plus(Int(1), Int(2)), Plus(Int(3), Int(4)))")	EInt 10
Reflect(Str("Assign(\"y\", Int(5))"))	EInt 5

Chapter 16

Taint Analysis

"dirty" is Dirty, "clean" is Clean

INPUT	VALUE
"dirty"	Dirty
"clean"	Clean
e	Equals(Plus(Val("x"),Val("y")),Int(6))
assign1	CSeq(Assign("x", Val("dirty")), Assign("y", Val("clean")))
assign2	CSeq(Assign("x", Val("clean")), Assign("y", Val("dirty")))
d	DSeq(New("x", Val("dirty")), New("y", Val("clean")))
c	CIfThenElse(e, assign1, assign2)

INPUT	OUTPUT
$\text{taint}_{\text{analysis}}$ Prog(d,c)	[("clean", TLoc 15n); ("dirty", TLoc 27n); ("x", TLoc 76n); ("y", TLoc 41n)] [(15n, Clean); (27n, Dirty); (41n, Dirty); (76n, Dirty)]