giaco.ml

an interpreted EDSL written in OCAML

Mario A. Barbara
Andrea Cancellieri

1 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.

2 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
- due to the non-recursive nature of `Procedure` and `Call`, they have been omitted from reflection and taint-analysis. The syntax restriction is due to the difficulty of implementing a parser for the complete OCAML list syntax, and the static analysis of commands that cannot be recursively unpacked (one lies in the *expressions*, the other in the *commands*)

3 Usage

the interpreter can be loaded up with

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

and subsequent testing may be performed:

```
>> (* have a functional sample: *)
>> Plus(Int(3), Int(4)) in eval e emptyenv emptystore;;
- : evalue = EInt 7

>> (* now let's try a complete program *)
>> let e = Lambda("x",Multiply(Val("global"), Var("x"))) in
>> let d = New("global", Str("yes")) in
>> let c = Assign("global", Apply(e, Int(5))) in
>> let program = Prog(d,c) in
>> let result_environment, result_store = interpret' program emptyenv emptystore in
>> eval (Val("global")) result_environment result_store;;
- : evalue = EStr "yesyesyesyesyes"
```

The most useful interpreter functions are:

`eval: expr -> env -> store -> evalue` evaluates expressions

`cval: com -> env -> store -> store` converts commands into a modified mutable memory

`dval: dec -> env -> store -> env*store` allows extending mutable memory

`interpret: prog -> env -> store -> store` combines all of the above. cannot return expression values like eval, though, as we do not have print functionality

`interpret': prog -> env -> store -> env*store` just like `interpret` but return the last evaluated environment as well, which allows for further analysis of the output

`emptyenv` and `emptystore` already initialized empty environment and store

`env'` and `store'` allow extending environments and stores outside of syntax. `new'` combines this and allows doing something like the `New(...)` command outside of syntax.

`ereflect` and `creflect` and `dreflect` and `reflect` reflection of expressions, commands, declarations, full blown programs.

`etaint` and `ctaint` and `dtaint` and `taint_anlaysis` static taint analysis of expressions, commands, declarations and full blown programs.

3.1 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()`