Flix and its Implementation:
A Language for Static Analysis

Ming-Ho Yee, Magnus Madsen, Ondřej Lhoták

UNIVERSITY OF WATERLOO
FACULTY OF MATHEMATICS
David R. Cheriton School
of Computer Science

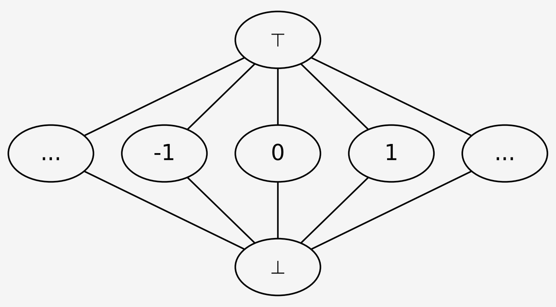
Introduction

Flix is a language for implementing static analyses. Flix is inspired by Datalog, but supports user-defined lattices and functions, allowing a larger class of analyses to be expressed. For example, a constant propagation analysis can be expressed in Flix, but not in Datalog.

A static analysis in Flix is specified as a set of constraints in a logic language, while functions are expressed in a pure functional language.

Constant Propagation Analysis

```
enum Constant {
  case Top,
  case Cst(Int),
  case Bot
}
```




```
def leq(e1: Constant, e2: Constant): Bool =
  match (e1, e2) with {
    case (Bot, _)            => true
    case (Cst(n1), Cst(n2)) => n1 == n2
    case (_, Top)            => true
    case _                   => false
  }

def sum(e1: Constant, e2: Constant): Constant =
  match (e1, e2) with {
    case (_, Bot)            => Bot
    case (Bot, _)            => Bot
    case (Cst(n1), Cst(n2)) => Cst(n1 + n2)
    case _                   => Top
  }

rel AsnStm(r: Str, c: Int)
rel AddStm(r: Str, x: Str, y: Str)

lat LocalVar(k: Str, v: Constant)

LocalVar(r, Cst(c))      :- AsnStm(r, c).
LocalVar(r, sum(v1, v2)) :- AddStm(r, x, y),
                            LocalVar(x, v1),
                            LocalVar(y, v2).
```

Fixed-Point Semantics

Ex 1. *Input Facts*

```
AsnStm("a", 40).        // a = 40
AsnStm("b", 2).         // b = 2
AddStm("c", "a", "b").  // c = a + b
```

Minimal Model

```
LocalVar("a", Cst(40)).
LocalVar("b", Cst(2)).
LocalVar("c", Cst(42)).
```

Ex 2. *Input Facts*

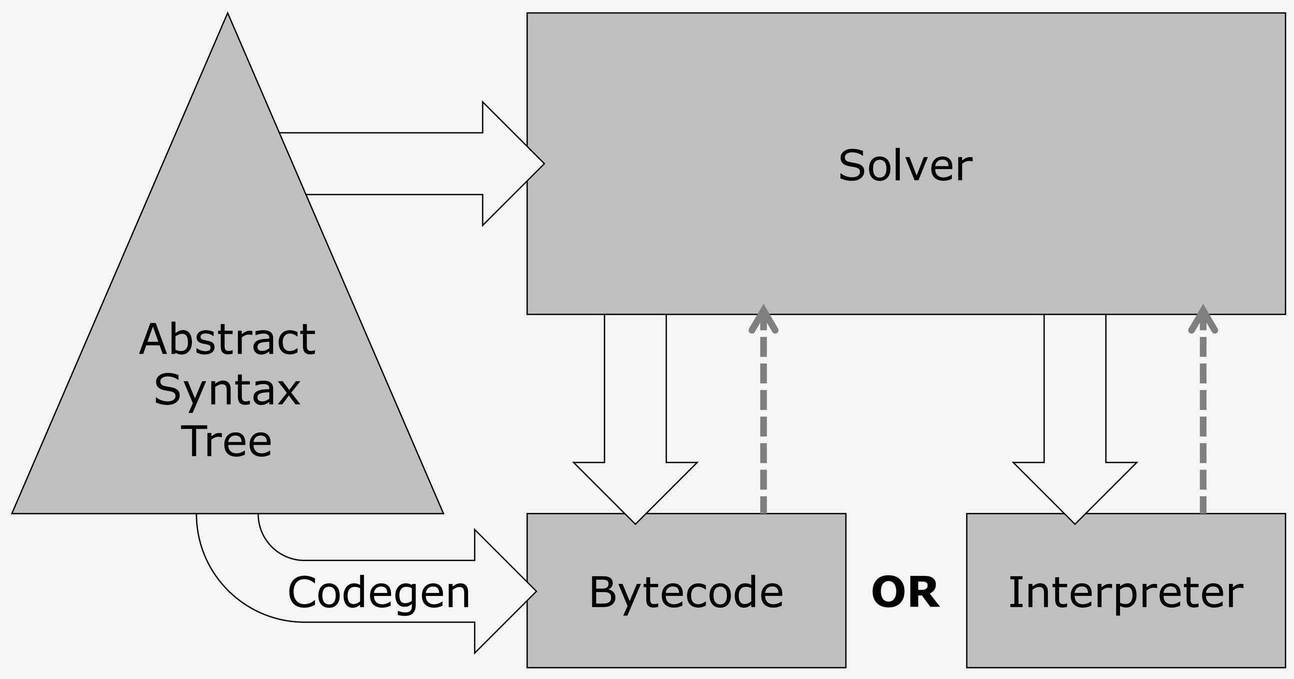
```
AsnStm("a", 40).        // a = 40
AsnStm("b", 1).         // b = 1
AsnStm("c", 2).         // c = 2
                        // if (mystery())
AddStm("d", "a", "b").  //    d = a + b
                        // else
AddStm("d", "a", "c").  //    d = a + c
```

Minimal Model

`LocalVar("a", Cst(40)).`
`LocalVar("b", Cst(1)).`
`LocalVar("c", Cst(2)).`
~~`LocalVar("d", Cst(41)).`~~
~~`LocalVar("d", Cst(42)).`~~
`LocalVar("d", Top).`

Back-end Architecture

The solver evaluates the logic language while the interpreter evaluates the functional language.



Compiling to JVM Bytecode

To improve performance, we compile the functional language to JVM bytecode and replace the intepreter.

Desugaring pattern matches

A pattern may involve equality checks, bind values to variables, or contain subpatterns. The cases are processed individually and then linked together.

```
match x with {
  case PAT1 => EXP1
  case PAT2 => EXP2
  case PAT3 => EXP3
}
```

```
let v' = x in
let err = λ() ERROR in
let e3  = λ() if (PAT3 succeeds) EXP3 else err() in
let e2  = λ() if (PAT2 succeeds) EXP2 else e3()  in
let e1  = λ() if (PAT1 succeeds) EXP1 else e2()  in
  e1()
```

Implementing lambdas

We eliminate free variables through closure conversion and lambda lifting.

```
def f(a) = let g = λ(x, y) a+x+y in
  g(1, 2)

// after closure conversion
def f(a) = let g = MkClo(λ(a', x, y) a'+x+y, a) in
  g(1, 2)

// after lambda lifting
def f$0(a', x, y) = a'+x+y
def f(a) = let g = MkClo(f$0, a) in
  g(1, 2)
```

`MkClo` is compiled to an InvokeDynamic call to create a Java 8 lambda, via `java.lang.invoke.LambdaMetafactory`. Lambdas are invoked through interface calls.

This research was supported by the Natural Sciences and Engineering Research Council of Canada.

flix.github.io