Programming a Dataflow Analysis in Flix

[Tools for Automatic Program Analysis (TAPAS), 2016]

Magnus Madsen
University of Waterloo
mmadsen@uwaterloo.ca

Ming-Ho Yee
University of Waterloo
ming.ho-yee@uwaterloo.ca

Ondřej Lhoták
University of Waterloo
olhotak@uwaterloo.ca

ABSTRACT

Flix is a logic and functional language designed for the implementation of static analysis tools. Flix is inspired by Datalog and extends it with user-defined lattices as well as monotone filter and transfer functions. In recent work, Flix has been used to express several analyses, including the Strong Update analysis, and the IFDS and IDE algorithms.

1. INTRODUCTION

Designing, implementing, and testing static analysis tools is a challenging and complex task. The designer is often faced with difficult trade-offs between trying to ensure soundness, precision, and overall scalability of the analysis. To overcome these challenges, and to simplify the implementation, some designers have turned to Datalog [2].

Datalog is a logic programming language for expressing *constraints on relations*. A Datalog program is a collection of facts and rules. Each rule can derive new facts from the existing facts, and this process is iteratively repeated until the least fixed point of the rules is found. In past work, Datalog has successfully been used to specify large-scale points-to analyses of object-oriented programs, in particular for Java [1, 4]. Datalog is not limited to points-to analyses, but it can also be used to express other analyses that are based on constraints on relations, e.g. definite assignment, reaching definitions, and available expressions.

However, many static analyses, including classic dataflow analyses that have been known for over thirty years, are not defined as constraints on *relations*, but as constraints on *lattices*. These include sign analysis, constant propagation analysis, and interval analysis. Regrettably, such analyses cannot be expressed by Datalog.

We have proposed Flix to overcome these limitations [3]. Flix is inspired by Datalog and extends it with user-defined lattices as well as monotone filter and transfer functions. With Flix, we can implement static analyses that are difficult, or even impossible, to express in Datalog.

2. FROM DATALOG TO FLIX

A Datalog program consists of rules of the form:

$$H(\bar{t}) \Leftarrow B_1(\bar{t}), \ldots, B_n(\bar{t}).$$

where H and each B_i are predicate symbols, $\bar{t}$ is a sequence of terms, and a term t is either a variable or constant. Intuitively, if the *body* predicates $B_1, \ldots, B_n$ in a rule are satisfied then the *head* predicate H of the rule must be satisfied.

In a Flix program, a rule has the more general form:

$$H(\bar{t}, f(\bar{t})) \Leftarrow \varphi(\bar{t}), B_1(\bar{t}), \ldots, B_n(\bar{t}).$$

and each predicate symbol is (optionally) associated with a lattice. In the rule, φ is a *monotone filter function* and f is a *monotone transfer function*. We express such functions as *code* in the functional language of Flix. Crucially, this allows functions to have infinite domains/codomains, something which is not possible in Datalog. To illustrate the role played by lattices in Flix, consider the Datalog program:

```
A("foo").    A("bar").    B("qux").
```

this program has the unique solution, i.e. minimal model:

$$\{A(\texttt{"foo"}), A(\texttt{"bar"}), B(\texttt{"qux"})\}$$

whereas the Flix program:

```
A(Cst(1)).    A(Cst(2)).    B(Cst(3)).
```

where A and B are defined over the *constant propagation lattice*, and `Cst(c)` is the constructor for the constant c, has the minimal model:

$$\{A(\top), B(\texttt{Cst}(3))\}$$

Notice how the two facts $A(\texttt{Cst}(1))$ and $A(\texttt{Cst}(2))$ are combined into the fact $A(\top)$ according to the lattice order.

3. DATAFLOW ANALYSIS

We now show how to express a simple dataflow analysis in Flix. We will implement an intra-procedural *constant propagation* analysis. The purpose of this analysis is to compute, for every local variable, whether it points to a single constant, and if so, the value of that constant. We begin by defining three *input relations*:

```
rel LitStm(r: Var, c: Int)
rel AddStm(r: Var, x: Var, y: Var)
rel DivStm(r: Var, x: Var, y: Var)
```

The three relations represent the program-under-analysis. Specifically, the first relation represents literal statements of the form $r = c$, where c is an integer literal and r is the result variable; the second relation represents addition statements of the form $r = x + y$, where r is the result variable and x and y are the operands; and finally, the third relation represents division statements of the form $r = x/y$, where r is the result variable, x is the numerator, and y is the denominator. Next, we define the lattice

```
lat LocalVar(k: Var, v: Constant)
```

This lattice is the result of the analysis: it maps every local variable to an element of the constant propagation lattice.

In order to implement this lattice, we must define the type of its elements, a partial order on those elements, and the usual lattice operations. We begin by defining the `Constant` type as an algebraic data type with three variants:

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

Here, `Bot` and `Top` represent the bottom and top elements of the lattice, respectively, and `Cst`(c) represents the specific constant c. The next step is to define the *partial order*, as a function, on the elements of this lattice:

```
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
  }
```

This function takes two arguments `e1` and `e2`, which are lattice elements, and returns `true` if `e1` is smaller than or equal to `e2`. The function pattern matches on its arguments and returns `true` if the first argument is bottom, if both arguments are the same constant, or if the last argument is top. Otherwise it returns `false`. We define functions for the *least upper bound* and *greatest lower bound* in a similar way.

We are now able express the semantics of the analysis using three simple rules, one for each of the input relations:

```
LocalVar(r, Cst(c)) :- LitStm(r, c).

LocalVar(r, sum(v1, v2)) :- AddStm(r, x, y),
                            LocalVar(x, v1),
                            LocalVar(y, v2).

LocalVar(r, div(v1, v2)) :- DivStm(r, x, y),
                            LocalVar(x, v1),
                            LocalVar(y, v2).
```

The first rule states that if there is a literal statement $r = c$, where the variable r is assigned the constant c, then we simply assign r the constant propagation lattice element `Cst(c)`. The second rule states that if there is an addition statement $r = x + y$, where the value of the local variable x is v_1 and the value of y is v_2, then the value of r is at least the result of applying the `sum` function to v_1 and v_2. The third rule is similar, but for division statements.

The functions `sum` and `div`, referred to in the head of the last two rules, are *transfer functions* expressed in the functional language of Flix. Here is the definition of `sum`:

```
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
  }
```

This function takes two arguments. If either is bottom then the result is bottom. If both are constants then the result is the sum of the constants. Otherwise, at least one argument is top, thus the result is top.

This completes the implementation. The analysis computes an element of the constant propagation lattice for every local variable in the program-under-analysis.

We can easily extend the analysis to find bugs, for instance, detecting potential division-by-zero errors. To do so, we introduce a relation to capture result variables that are possibly indeterminate

```
rel ArithmeticError(r: Var)
```

and we introduce the rule:

```
ArithmeticError(r) :- isMaybeZero(y),
                      DivStm(r, n, d),
                      LocalVar(d, y).
```

This rule states that if there is a division statement $r = n/d$, where the value of the denominator variable d is y and y is possibly zero according to the *filter function* `isMaybeZero`, then the result variable r is possibly indeterminate. Here, `isMaybeZero` is the function

```
def isMaybeZero(e: Constant): Bool
  = match e with {
      case Bot    => false
      case Cst(n) => n == 0
      case Top    => true
  }
```

This function takes one argument and returns `true` iff it is either the zero or top element.

4. CORRECTNESS

Every Datalog program eventually terminates and returns the minimal model. Flix programs, on the other hand, may fail to terminate if the user-defined lattices, filter, and transfer functions fail to satisfy a range of mathematical properties, e.g. monotonicity. To ensure termination and the existence of a minimal model, we have implemented a *verifier* for Flix programs. The verifier, which is based on symbolic execution and satisfiability modulo theories, checks that the program satisfies the required properties or outputs an error message along with a counter-example.

5. SUMMARY

Flix is a logic and functional programming language designed for the implementation of static analyses. Flix is inspired by Datalog and extends it with user-defined lattices and monotone filter and transfer functions. Using Flix, static analysis implementors can express a broader range of analyses than is possible in pure Datalog, while retaining the familiar rule-based syntax of Datalog.

Flix is open-source and freely available on GitHub:

http://github.com/flix/

More information can be found on the Flix website:

http://flix.github.io/

6. REFERENCES

[1] M. Bravenboer and Y. Smaragdakis. Strictly Declarative Specification of Sophisticated Points-To Analyses. In *Proc. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2009.

[2] S. Ceri, G. Gottlob, and L. Tanca. What You Always Wanted to Know About Datalog (and Never Dared to Ask). *Transactions on Knowledge and Data Engineering*, 1989.

[3] M. Madsen, M.-H. Yee, and O. Lhoták. From Datalog to Flix: A Declarative Language for Fixed Points on Lattices. In *Proc. of the 37th Conference on Programming Language Design and Implementation (PLDI)*, 2016.

[4] Y. Smaragdakis and M. Bravenboer. Using Datalog for Fast and Easy Program Analysis. In *Datalog Reloaded*, 2011.