COMP90053: Programming Project 2013 #1

Due on Friday, May 24, 2013

Program Analysis and Transformation

Diana Barreto, Ivan P. Valarezo id:574386 - id:601099

An Interval Analysis for Tip

Introduction

Abstract Interpretation allows to understand and obtain information about possible behaviours of a program without executing it. This report presents the design of an example implementation of abstract interpretation using data flow analysis (representation of the program using a graph) to know the possible values that could take the variables in each program point represented by an interval. Therefore the lattice used to implement the abstract interpretation is the interval lattice.
The implementation was develop using Haskell due to the facilities that this language allows to implement mathematic expressions the programs that can be analysed are the ones that could be generated using a language called TIP.

Task Solution

After reviewing the task specifications and information about how to perform the task[1]. The program was split in the following parts:

- *Tip Parser (TParse, Syntax)*: This component was built using Happy[1] and the modules previously provided for this project.
- *Tip Flow Generator (TCFlow)*: These modules convert a tip program in a list of CFGNode(s).
- *Computation Sequence (TControl)*: This module finds the predecessors of each node of the list of nodes.
- *Interval Operations (TInterval)*: In this module the arithmetic, union and intersect operations are defined for intervals.
- *Evaluation of Interval Expressions (TEvalInterval)*: This module includes the functionality to transform TIP expressions on interval expressions, additionally it includes the functionality to evaluate interval expressions.
- *Set of VarState operations (TVarStateOperations)*: This module contains functions to manipulate a set of variables, and to calculate the new values of the variables.
- *The Interval Analysis(Main)*: The main module in charge of the interval analysis and widening/narrowing.

Tip Parser

The parsing has been adapted starting from the module provided for this project, done using Happy Parser Generator, this package has allowed us to start with a set of well defined syntax elements to be used for our subsequent phases.

Flow Generator and Control Representation

The first part of this module defines the data type to hold the graph, a linear simple structure based on nodes:

```
data CFGNode
    = AsgNode String Exp
    | OutputNode Exp
```

[1] http://www.haskell.org/happy/

```
    | GotoNode Int
    | IfGotoNode Exp Int
    | EntryNode
    | ExitNode
    deriving (Show,Eq)
```

Since we will be doing a forward analysis, this simple structure will help us to produce the set of nodes and predecessors (based on some ideas from [3] and [2]).

Computation sequence

After getting the linearized nodes structure, this step will traverse the graph to get a collection of predecessors, this step is important since with this new structure, this process will provide valuable information to be feed to the interval analysis. The Haskell structure to hold this information is represented as follows:

```
type PredCFGNode = (CFGNode,[Int])
```

The *Int* list holds the node id references for each *CFGNode* from where this node could be reached. The operations included in this module, are also in charge of constants collecting, to be used for the widening phase.

Interval Operations

The interval operations of this module has been constructed overloading the Haskell type classes for the Ord and Num. Haskell incorporates a strong extendible type scheme, from where we have shaped the data logic and the operations behaviours. We have created and overloaded the following operations:

$\leq$ less than or equal

$>$ greater than

$+$ interval addition

$-$ interval difference

$*$ interval multiplication

$\div$ interval division

$\cap$ meet

$\cup$ join

Evaluation of Interval Expressions

In order to evaluate the program expressions using an Interval Domain, two functions were developed and are important to mention:

transformExp: The data type of the tip programs are mainly integers (concrete domain), however in the interval analysis is used the abstract domain of interval. For this reason was created this function that transform Integer expressions in Interval Expressions.

evalInterExp: This function receives an Interval expression, with the corresponding variable replaced by an interval and calculates the expression, yielding an Interval as a result. In the case of an expression of type a > b, the result is [1,1] if the condition is *True* for all interval numbers. The result is [0,0] if the condition is *False* for all interval numbers and Finally if the condition is True for some numbers and False for others the result of this expression is $[-\infty, \infty]$.

Set of VarState operations

The set of *VarState* operations include auxiliary functions to perform the algorithm to do the interval analysis. It include for example functions to know the intervals that represent a condition, functions to initialize states (bottom or top), functions to union and intersect *VarStates* , used to resolve the equations in each program point.

The Interval Analysis

The main data structures involved in the interval analysis are:

- *type VarState = [(VarName, Interval)]* : This represents the possible values of a set of variables in a program point of a tip program.
- *type VarStates = [VarState]* : This represents the possible values that could take the set the variables in the different program point as a result of kleene iteration.
- *type PredCFGNode = (CFGNode,[Int])* : A collection of nodes and predecessors (in a tuple)

The Interval Analysis is performed using three main functions:

iteration: This function represents the execution of a kleene iteration, in other words this if the function $f(x_1, x_2, ..., x_n)$ that will be executed n times in order to find the fix point. Consequently this function receives a VarStates with the result of the previous execution and returns a new VarStates with the result of the current iteration.

Additionally, to the *VarStates* of the previous iteration, this function receives a *PredCFGNode* list. The algorithm goal is to process each element of this list. To fulfil this task structural induction based in *CFGNode* is used, it means that a program equation was developed for each data constructor of the type *CFGNode* and the non recursive equation or exit case correspond to the data constructor "Exit Node".

According to the Data Constructor of the node, the algorithm perform the following steps:

1. **Union of Predecessors**: Find the values of the set of variables in the predecessors nodes and union them. This values are obtained for the current state if the predecessor has been already evaluated other case they are obtained from the previous state.
2. **Intersection of condition**: If the node is a conditional node it is possible to restrict the values according with the condition. The intervals for intersections are calculated considering the two branch of a condition True and False. The False conditional interval is pass to the goto node that represent the false branch to obtain the value of the variables in this point if the restriction is false.
3. **Calculation of new value**: The union values are intersect intervals of the condition and this new value is added to the new state.
4. **Evaluate Reachable**: Considering the current values of the variables it evaluate the program points that the program could arrive.

wideningState: This function do the work of widening. This receives two VarStates and go over the two *VarStates* in a recursive way, getting the pair of variables and applying the widening operator to generate a new *VarStates* with the result.

iterations: This function control the process to obtain a fix point. To do this task the function invoke iterations and *wideningState* . The function first call iteration passing as a previous *VarState* the variables initialized in *Empty* . Then the process invoke the function *i times* , but before invoke each iterations the algorithm compare if the previous *VarState* is equal to new one if it is the case means that it is already stabilized, this will be the approximation returned. Otherwise it applies widening over the two last *VarStates* calculated and the result is the entry to execute iteration *i times* . In this way it is calculated the variables values using widening and then the function is executed again, having as a result narrowing.

Dead code elimination Module

The Optimizer module eliminates dead code (Code marked by the Interval Analysis as unreachable), and presents a simplified version of the Tip code. For example:
Given a program like this:

```
i=9;
if (i>10){
    j=12;
    k=13;
}else{
    j=0;
    k=0;
}
i=12;
```

Our production algorithm generates this linearization:

```
0:<entry>
1:i = "9"
2:if "i > 10" goto 4
3:goto 7
4:j = "12"
5:k = "13"
6:goto 9
7:j = "0"
8:k = "0"
9:i = "12"
10:<exit>
```

The widening and interval analysis generates this:

```
0                     | 0:
   <entry>            |    <entry>
1                     | 1: i=[-oo,oo] j=[-oo,oo] k=[-oo,oo]
   i = 9              |    i = 9
2                     | 2: i=[9,9] j=[-oo,oo] k=[-oo,oo]
   if i > 10 goto 4   |    if i > 10 goto 4
```

```
3                    | 3: i=[9,9] j=[-oo,oo] k=[-oo,oo]
   goto 7            |    goto 7
4                    | 4: i=_|_ j=[-oo,oo] k=[-oo,oo]
   j = 12            |    j = 12
5                    | 5: i=_|_ j=_|_ k=_|_
   k = 13            |    k = 13
6                    | 6: i=_|_ j=_|_ k=_|_
   goto 9            |    goto 9
7                    | 7: i=_|_ j=_|_ k=_|_
   j = 0             |    j = 0
8                    | 8: i=_|_ j=[0,0] k=[-oo,oo]
   k = 0             |    k = 0
9                    | 9: i=_|_ j=[0,0] k=[0,0]
   i = 12            |    i = 12
10                   | 10: i=[12,12] j=[0,0] k=[0,0]
   <exit>            |    <exit>
```

. . . And, the resulting code should be improvised like this:

```
0:<entry>              |0:<entry>
1:i = 9                |1:i = 9
2:if i > 10 goto 4     |2:k = 0
3:goto 7               |3:i = 12
4:j = 12               |4:<exit>
5:k = 13               |
6:goto 9               |
7:j = 0                |
8:k = 0                |
9:i = 12               |
10:<exit>              |
```

Our algorithm has these well defined tasks:

- Remove the dead code marked as 'NR' from the interval analyzer.
- from the previous step, we obviously will break some *Goto* statements, so the next step is to filter also this brocken statements.
- filter some consecutive goto nodes ie. nodes that are pointing to the next lines in the code without any obvious advantage.
- finally, re arrange the nodes, re enumerating the still alive "if" and "goto" sentences and produce a meaninful output.

Fixes to the previous version

Some fixes done to our previous project:

- Fixed the interval division, improved by splitting intervals and operating over this subsets. The main idea is to avoid to deal with divisions by 0.

- Created a new structure for wrapping Interval and allows us to mark non reachable branchs of code.
- In order to better handling some interval operations, we have added a new data type (SInt). And a set of conversion functions.

Most of the Interval code takes advantage of the *Haskell* deriving feature to make our types behave in the intended way. At most, the *instances* are derived from the following *Class* types:

Show: for presentation

Eq: for Bound and Interval evaluation

Num: for arithmetic Bound and Interval operations

Fractional: for Interval division

Also, the Join (union) and Meet (intersect) operations where fixed.

Conclusion

During the design phase is always important to think about what is the best data structure that could suit the given problem, in our case, a not so optimal decision was to use an Interval data type with non related parts (the lower bound Lb and the Upper bound Ub). This decision complicated greatly the interval operations since all cases had to be analyzed independently and most of the cases we where stuck between Lb against Ub operations. To this complexity we also faced the added complication of dealing with $-\infty$ and ∞ as operands, that in this scheme complicated even more the overall interval operations.
Finally, the solution iterates correctly using the Kleene algorithm and helped us to understand the main purpouse of this taks.

User Guide

To use the program a compilation thru *ghc* is required, the procedure is as follow:

```
$ ghc --make Main.hs -o hskia
```

The previous step should generate a *hskia* executable file,

```
$ ./hskia
Usage: hskia [-i -p -a -o] filename

      -a: get the production list from file

      -p: get the predecesors list from file

      -i: start the interval analysis

      -o: optimize the code given in filename
```

... where:

-a Prints the productions for the tip program in filename, this does not include the interval analysis.

-p Prints the predecesors list for debugging the program analysis later, this option also does not include the interval analysis.

-i Starts the interval analysis, including widening and generates the analysis report.

-o Creates an optimized version of the tip program <filename>, it includes the interval analysis

References

[1] Schwartzbach, I. , Lecture Notes on Static Analysis. University of Aarthus, Denmark. misbricks.dk.

[2] Bourdoncle, F. (1993, January). Efficient chaotic iteration strategies with widenings. In Formal Methods in Programming and their Applications (pp. 128-141). Springer Berlin Heidelberg.

[3] Cousot, P., & Cousot, R. (1977, January). 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 (pp. 238-252). ACM.