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 all 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 developed 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, meet(union) and join(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.
- *Conditionals Constraints (TEvalConditions)*: This module includes the functions to evaluate the constraints that should be added to the variables in the conditionals statements.
- *The Interval Analysis(Main)*: The main module in charge of the interval analysis and widening/narrowing.

Tip Parser

The parsing provided for this project was developed using Happy Parser Generator. this package has allowed to start with a set of well defined syntax elements to be used in the 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:

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

```
data CFGNode
    = AsgNode String Exp
    | OutputNode Exp
    | GotoNode Int
    | IfGotoNode Exp Int
    | EntryNode
    | ExitNode
    deriving (Show,Eq)
```

Since interval analysis is performed using forward analysis, this simple structure is useful to produce the set of nodes and predecessors (based on some ideas from [3] and [2]).

Computation sequence

After getting the linearized nodes structure, the next step is to traverse the graph to get a collection of predecessors. This is important since this allows to generated a new structure that provides valuable information to feed the interval analysis. The Haskell structure that represents this information is composed for a node and a list of Int that contains the id references of each CFGNode from where this node could be reached. The structure is represented as follows:

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

An additional responsibility of this module is to include the operations for collecting the constants (landmarks) to be used in 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 have been shaped the data logic and the operations behaviours. The operations created and overloaded are the following:

$\leq$ less than or equal

$>$ greater than

$+$ interval addition

$-$ interval subtraction

$*$ interval multiplication

$\div$ interval division

$\cap$ meet (union)

$\cup$ join (intersection)

Evaluation of Interval Expressions

In order to evaluate the program expressions using an interval domain, two main functions were developed and it is important to mention them:

transformExp: The tip language have been designed to deal with integers (concrete domain), however in interval analysis is used the abstract domain of intervals. For this reason was created this function that transform Integer expressions in interval expressions. To perform this task the function follows the patterns given by the structure of integer expressions and as a result that the interval expression data structure is a mirror of the integer one, the conversion is match one pattern again the other.

evalInterExp: This function receives an interval expression where the values of the variables have been previously replaced by an interval. It evaluates the expression using interval operations, yielding an Interval as a result. for example it evaluated [0,0]+[1,1] and the result is [1,1]. In the case of conditional Interval expression (type a > b or a == b) the result of evaluating the expression is [1,1] if the condition is only *True* for all interval values. Then the result is [0,0] if the condition is *False* for all interval numbers and finally if the condition is true for some values and false for others the result of this expression is [0,1].

Set of VarState operations

The set of *VarState* operations includes auxiliary functions to perform the algorithm to do the interval analysis. It include for example functions to initialize states (bottom or top), functions to union and intersect *VarStates* which are used to resolve the equations in each program point and functions to manage to obtain and replace values in a *VarState* and in a *VarState*.

Conditionals Constraints

This module calculates the constraints that should be applied, when a node with a conditional expression is obtained in order to have more precise approximations. In the program are considered two types of constraints. First the constraints that are resulted of an expression of type bigger than (>) in this case the constraints are intersect with the value of the variables at this program point to obtain more refined results for the True and False branch. For example if the condition is (j>5) it is possible to say that if the condition is true the values of j should fall in the interval [6,∞] and if it is false the constraint interval should be [−∞,5].
The second type of constraints are the ones obtained for the expressions of type (==) in this case for the true branch the constraint is also intersected, however for the false branch the value is optimized if the result of the constraint allows to remove border values, for example, if the condition is j==[0,5] and the value of the variable is [0,10] it is possible to remove the border to obtain a more refined value of [5,10] for the false branch.

The Interval Analysis

The main data structures involved in the interval analysis are:

- *data AbsValue= NoReach | AInterval* : This is a wrap for an Interval to differentiate the value when a variable is bottom(no initialized) or no reachable.
- *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 their 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 fixed 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 and 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 corresponds to the data constructor "Exit Node". Each time that a node is evaluated a new *VarState* is added to the list of *VarStates* that corresponds to the state of the current iteration. The program also manages a list of reachable points that allows to each node to know if it is a reachable point or not.

For each node in the list the algorithm executes the following steps to decide the *VarState* or value of the variables in each program point for the current iteration:

1. **Evaluate Reachability**: The first thing that it is evaluated in any kind of node is if it is a reachable point (the element exists in the list of reachable). If it is not reachable, so it is added a *VarState* in the current iteration with all variables with value NR (No Reachable) and nothing is added to the list of not reachable, but if the node is reachable the successors are added to the list of reachable and the following steps are executed (in a conditional node the successor depends on if the condition is True, False or Both).
2. **Union of Predecessors**: This step applies to all kind of nodes and consists in to find the values of the variables in the predecessors nodes and applying union to all of them. If the value of a predecessor node has not been calculated in the current iteration, so the state with the past values is used to obtain them.
3. **Intersection with Condition Constraints**: If the node is a conditional (it means the type "IfGotoNode Exp Int") node, so it is possible to restrict the values of the intervals according with the constraints. To do this the result of the union of the predecessors variables is refined intersecting with constraints values. It is important to notice that the information of the constraints is obtained in this point but the information is passed through the program calls to the True and False branch to improve the precision of the intervals in those future program points.

wideningState: This function does 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* as a result.

iterations: This function controls the process to obtain a fixed point. To do this task it invokes *iteration* and *wideningState* functions. First it calls iteration passing *VarStates* with the variables initialized in *Empty* (bottom) as an initial old state. The process invokes this function *i times* , but before invokes each iteration the algorithm compares if the previous *VarStates* is equal to the new one and if it is the case means that it is already stabilized and this approximation is returned as the final result. Otherwise when it achieves i iterations it applies widening over the two last *VarStates* calculated. This result is the entry to begin a cycle of other *i times* iterations, which means that narrowing is being applied. To guarantee the end of the program, the process mentions above is executed *j times* and after that a final approximation is returned.

Dead code elimination Module

The Optimizer(TOptimizer) module eliminates dead code (Code marked by the Interval Analysis as unreachable). For example:
Given a program like this:

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

The 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 output:

```
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=NR j=NR k=NR
   k = 13              |    k = 13
6                      | 6: i=NR j=NR k=NR
   goto 9              |    goto 9
7                      | 7: i=NR j=NR k=NR
   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 improved 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>               |
```

The algorithm has these well defined tasks:

- Remove the dead code marked as 'NR' from the interval analyser.
- From the previous step were removed some "goto" nodes, so the next step is also to filter the statements that invoked those "goto" nodes.
- Filter some consecutive "goto" nodes. It means, nodes that are pointing to the next lines in the code without any advantage.
- Finally, re arrange the nodes, re enumerating the still alive "if" and "goto" sentences to produce a meaningful output.

Fixes to the previous version

Some improvements done to the previous project:

- Fixed the interval division, It was improved by splitting the intervals in positive and negative values and operating over this subsets. The main purpose is to avoid to deal with divisions by 0.
- Created a new structure for wrapping Interval to mark non reachable branches of code and to differentiate no Initialized from no Reachable.
- In order to improve the handling of some interval operations, the data type (data SInt = SMinInf | SInt Int | SPlusInf) and a set of conversion functions were added.
- The Join (union) and Meet (intersect) operations were fixed.
- The conditions for the case false of the conditional (==) were added.
- The neighbours (adding and subtracting one) of the landmarks were added to obtain more precise results.
- The Entry point was modified to initialized the variables in bottom instead of top.

Most of the Interval code takes advantage of the *Haskell* deriving feature to make types behave in the intended way. The majority of 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

Lessons Learned

After finishing this project the main lesson learned is that during the design phase is always important to think about what is the best data structure that could suit in a given problem, in our case, a not optimal decision was to use an Interval data where the values do not belong to the same data type (was used lower bound Lb and the Upper bound Ub in separated way). This decision complicated the interval operations since all cases had to be analysed independently and in many cases we have difficulties to combine operations with Lb and Ub and $-\infty$ and ∞. It is also important to try to design an extensible and maintainable code to facilitate future improvements. Finally, This project allows to understand more clearly the concepts of abstract interpretation specially interval analysis. Also it gives us an opportunity to know more about Haskell and, having the opportunity of fix some mistakes after a feedback allowed us to learn even more.

User Guide

To use the program a compilation *ghc* is required and 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 predecessors 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 predecessors 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.