Overapproximating the Cost of Loops

Master Thesis

Author(s):
Schweizer, Daniel

Publication date:
2013

Permanent link:
https://doi.org/10.3929/ethz-a-009767769

Rights / license:
In Copyright - Non-Commercial Use Permitted

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Overapproximating the Cost of Loops

Master's Thesis

Chair of Programming Methodology
Department of Computer Science
ETH Zurich

http://www.pm.inf.ethz.ch/

Daniel Schweizer, daschwei@student.ethz.ch

Supervisors: Dr. Pietro Ferrara, Prof. Dr. Peter Müller

April 9, 2013

Abstract

TouchDevelop is a novel programming language developed by Microsoft to write scripts on mobile devices. In this master's thesis we present a cost analysis that overapproximates the cost of loops in a TouchDevelop program. The analysis is implemented as an extension of Sample, a generic static analyzer based on the abstract interpretation theory. First, we run a numerical analysis relying on Apron on the input program. Then, we use the result of this analysis to infer a cost relation system for each loop in the program. Finally, this system is solved by the upper bounds solver PUBS. PUBS calculates a closed-form upper bound of the cost of each loop, which is used as the result of the cost analysis.

The implementation of the analysis has been deeply evaluated. First, we show its results on a series of input programs - including scripts written by ourselves and scripts from the TouchDevelop cloud - to illustrate its capabilities and limits. Then, to evaluate the performance of the loop cost analysis, we ran it on more than 1700 scripts from the cloud. The results show that the analysis scales up and is precise enough to be used in practice. For instance, we could use the cost information from the analysis at runtime to decide whether it is better to execute a script locally on the mobile device, or in the TouchDevelop cloud.

Acknowledgment

First, I would like to thank my supervisors Prof. Dr. Peter Müller and Dr. Pietro Ferrara. I am grateful to Prof. Müller for giving me the possibility to create this challenging Master's thesis at the Chair of Programming Methodology. A special thank goes to Pietro Ferrara for his excellent assistance.
I would also like to thank Lucas Brutschy for helping me with TouchDevelop and Sample. Further thanks go to Antoine Miné for providing the Apron library, and to Samir Genaim for providing the PUBS solver and for his helpful input in an early stage of this thesis. I would like to thank all the members of the Chair of Programming Methodology for their valuable discussion input at my thesis presentations. Finally, thanks go to my family and colleagues for supporting me during the past six months.

Contents

1 Introduction

The TouchDevelop, developed by Microsoft, is a novel programming language that gives users the possibility to write scripts on their mobile devices (in particular on the Windows Phone). It is particularly simple, since it must allow one to develop applications on mobile devices with limited screen size and input devices. TouchDevelop scripts are created by the users on their smartphones and executed within the TouchDevelop run time environment on the phone. These scripts can then be shared with other users by uploading them to the TouchDevelop cloud infrastructure. Because of the limited hardware resources of mobile devices, a cost analysis of a TouchDevelop script could provide some useful information to optimize its execution. In particular, we could attach cost information to the script and use this information at runtime to decide whether the application should be executed locally on the mobile device, or in the cloud.

The goal of this master thesis is to develop a cost analysis that correctly overapproximates the cost of all loops in TouchDevelop scripts. We implement the cost analysis in Sample, a powerful tool for static program analysis. This gives us the possibility to implement the cost analysis relying on numerical analyses (based on the Apron library) which already exist in Sample. We intend to use the result of the numerical analysis to find entry-exit relations and initial values for the variables appearing in a loop. Using this information, we can infer a system of cost relations. We then use PUBS to solve this system. PUBS is a tool to automatically obtain closed form upper bounds for cost relation systems. From PUBS we get an upper bound of the cost of the loop, which is the result of the cost analysis.

In chapter 2 of this thesis, we introduce TouchDevelop, with a focus on collections and control structures. In chapter 3, we formally define cost relations and cost relation systems. Furthermore, we introduce the cost model that we will use throughout this thesis. In chapter 4, we discuss the static analyzer Sample. In particular, we see what an analysis is, and how a program is represented. In chapter 5 we sketch the main contribution of this thesis by giving a step-by-step summary of our approach for inferring the cost of loops. The next two chapters are then a detailed description of our implementation of the approach. In chapter 6 we present a slightly modified version of Sample's TouchDevelop compiler on which our analysis relies. It mainly performs a pre-processing step needed by the loop cost analysis, which we call augmenting a control flow graph. In chapter 7, we present the core of the actual analysis. In particular, we explain how we use the result of a numerical analysis to infer a cost equation system which represents the cost of a loop. In chapter 8, we discuss the result of running our analysis on a set of case studies. This nicely illustrates the possibilities and the limits of our analysis. In chapter 9, we present the result of running our analysis on a selection of real scripts from the TouchDevelop cloud. In chapter 10, we discuss the experimental results that we obtained by applying the loop cost analysis on a large number of scripts from the TouchDevelop cloud. In chapter 11, we present some related work. In chapter 12 we conclude this thesis, and we sketch some possible extensions of our work.

2 TouchDevelop

TouchDevelop (TD) [9] is a novel development environment for the Windows Phone [16], which allows one to develop scripts directly on the smartphone. TouchDevelop allows its users to develop Windows Phone applications which can access the data (including sensor values) and the media on their phone. Furthermore, the applications can easily access cloud services, including storage, computing, and social networks. Most TouchDevelop scripts are rather small (typically less than 100 lines of code).

TouchDevelop scripts are developed on the Windows Phone and executed within the TouchDevelop run time environment on the phone. The scripts can then be shared with other users, by uploading them to the TouchDevelop cloud infrastructure [15]. The TouchDevelop cloud enables sharing of scripts, and it acts as a repository of all scripts developed and published by users. A user can create a script by copying and modifying an existing script from the TouchDevelop cloud. A script which is not such a modified version of an existing script is called a *root script*.

The TouchDevelop programming language is a typed, structured language, built around the idea of only using a smartphone's touch screen as input device when writing code. The language mixes imperative, object-oriented, and functional features. The API offers a number of predefined classes (representing basic data types and data structures, different kinds of media, and interfaces for the physical components of the device). Using a concept called *records* [14], the programmer can also define own tables and indexes (like in database systems) and create own objects (similar to other object-oriented languages with garbage collection).

2.1 Collections in TouchDevelop

The TouchDevelop API offers access to several types of collections. In contrast to other high level languages, the implementation of collections is completely hidden from the user.

Some kinds of collections are *mutable*. This means that a script can create such a collection, add elements to it, and remove elements from it. The API provides several types of mutable collections, e.g. *Number Collection*, *String Collection* or *Sprite Set*. Other kinds of collections are *immutable.* This means that the elements of the collection (e.g., a list of songs on the user's phone) can only be accessed, and it is not possible to add or remove elements. The API provides several types of immutable collections, e.g. *Songs* (which represents the collection of all songs on the user's phone), *Pictures* (all pictures on the phone) or *Contact Collection* (all address book entries on the phone).

2.2 Control Structures in TouchDevelop

TouchDevelop is a structured programming language. As such a language, it offers sequential composition, conditionals and three forms of loops as the only means to manage the control flow. In the remainder of this section, *condition* denotes a boolean expression, *collection* denotes a TouchDevelop collection (either mutable or immutable), *block, block*$_1$ and *block*$_2$ denote a sequential composition of statements, and *expr* denotes an arithmetic expression.

Conditional

A *conditional* in TouchDevelop has the usual form as follows:

`if` *condition* `then` *block*$_1$ `else` *block*$_2$

Loops

The three kinds of loops are the while loop, the foreach loop, and the for loop. The most generic kind of loop is the *while loop*, which has the following form:

`while` *condition* `do` *block*

The *foreach loop* iterates over all elements of a collection which are part of the collection when the execution of the loop starts. It has the following form, where *x* is a variable name:

`for each` *x* `in` *collection* `where` *condition* `do` *block*

This statement is equivalent to:

```
coll := collection.copy();
index := 0;
while index < coll->count do {
    x := coll.at(index);
    if condition then {
        block;
    }
    index := index + 1;
}
```

Note that a *foreach* loop operates on a copy of the collection. So adding to or removing elements from the collection in the body of a foreach loop has no effect on the copy of the collection over which the loop iterates, but effects the original collection.

The *for loop* uses an index variable *i* incremented from zero to *expr* - 1. The index variable is read-only, i.e., its value is increased by 1 after each loop iteration and can not be changed in any other way. It has the following form:

`for` $0 \leq i < expr$ `do` *block*

This statement is equivalent to:

```
coll := collection;
index := 0;
while index < coll->count do {
    x := coll.at(index);
    if condition then {
        block;
    }
    index := index + 1;
}
```

3 Cost Relations and Cost Model

Evaluating cost relations which are part of a cost relation systems plays a central role in this master's thesis. This chapter formally defines these concepts, based on [1]. Furthermore, we show in this chapter how we model the cost of a program in this thesis.

First, we need to introduce some basic definitions. In this chapter, we use *a* and *b* for integer numbers, *r* for rational numbers, and *n* for natural numbers. We use *v* and *w* to denote variables with an integer value. Given an expression *exp*, vars(*exp*) denotes the set of variables occurring in *exp*. The notation $\overline{v}$ refers to an ordered list of variables $v_1, \ldots, v_n$, for some $n > 0$. To keep things simple, in the following explanations we sometimes interpret these lists as sets.

3.1 Cost Relations

First, let us define the concepts of a linear expression and a linear constraint:

DEFINITION: A *linear expression* is a expression which has the form $a_0 + a_1v_1 + \ldots + a_nv_n$ for some $n \geq 0$.

We call a term a_iv_i in a linear expression a *variable-coefficient-pair*, and we call a_0 the *constant* of the linear expression.

DEFINITION: A *linear constraint* (or *linear relation*) has the form l_1 *op* l_2, where l_1 and l_2 are linear expressions, and *op* is a relational operator, $op \in \{=, \neq, <, \leq, >, \geq\}$.

We use *c* to denote a linear constraint. Note that linear constraints with rational coefficients can always be transformed to equivalent linear constraints with integer coefficients, e.g. $\frac{1}{3}x > y$ is equivalent to $x > 3y$. We use *P* and *Q* to denote sets of linear constraints of the form $\{c_1, \ldots, c_n\}$, which should be interpreted as the conjunction $c_1 \wedge \ldots \wedge c_n$.

Now we can introduce the notion of a cost expression:

DEFINITION: A *cost expression cexp* is recursively defined as follows:

- r
- $\mathrm{nat}(l)$
- $exp_1 + exp_2$
- $exp_1 * exp_2$
- exp^r
- $\log_n(exp)$
- n^{exp}
- $\max(S)$
- $exp - r$

where r is a non-negative rational number, n is a natural number, l is a linear expression, $\mathrm{nat}(a) = \max(a, 0)$, exp, exp_1 and exp_2 are cost expressions, and S is a non-empty set of cost expressions. Furthermore, for any assignment of the variables in vars(*cexp*), *cexp* must evaluate to a non-negative rational number.

Cost expressions are used to describe the closed-form upper bounds for the cost of loops, which we try to find in the scope of this thesis. Note that, by definition, they are always evaluated to non-negative values. Now we are able to define a cost relation and a cost relation system:

DEFINITION: A *cost relation* (or *cost equation*, respectively) is a guarded equation of the form $\langle C(\overline{v}) = exp + \sum_{i=1}^{k} D_i(\overline{w}_i),\ P\rangle$ with $k \geq 0$, where C and all D_i are symbols representing a cost, all variables $\overline{v} \cup \overline{w}_i$ are distinct variables, exp is a cost expression, and P is a set of linear constraints.

A cost relation $\langle C(\overline{v}) = exp + \sum_{i=1}^{k} D_i(\overline{w}_i),\ P\rangle$ defines the cost of $C(\overline{v})$ to be exp plus the sum of the cost of all $D_i(\overline{w}_i)$. We call the linear constraints in P the *guards* of the cost relation. They represent the conditions under which the equation can be evaluated, as well as size constraints for the variables in the equation.

DEFINITION: A *cost relation system (CRS)* is an ordered list of cost relations.

As an example, consider method *while* (listing 3.1). It contains a single while loop at program point 1. Listing 3.2 shows the cost relation system representing the cost of the loop in method while. We will discuss in section 8.4 how the loop cost analysis actually generates and solves the CRS. For now, we use this example only to show how a cost relation system is represented in the remainder of this thesis. As one can see in listing 3.2, we enumerate the cost equations in a cost relation, starting at 1, so we can easily refer to them in the discussion. While our implementation internally uses an elaborate system of cost symbols, in this written report we use the capital letters A, B, etc. for them to keep things simple (these symbols are just names, so it does not matter). Each cost relation consists of a left and a right part. The left part is the actual equation, which is the definition of some cost. The right part lists the guards of this cost relation, enclosed in curly brackets. In the example in listing 3.2, the first cost relation has `true` as its only guard, so it can always be evaluated. The second cost relation has 7 guards in total. All of them are by definition linear constraints.

```
action while(a: Number, b: Number) {
    $x := $a;
    $y := $b;
    $i := 1;
    while $i < $x + 2 * $y do {       // program point 1
        $i := $i + 1;
    }
}
```

Listing 3.1. Code of method *while*.

```
1) A(c1, a, b) = B(c1, a, b, a, b, 1)         {true}

2) B(c1, x, y, a, b, i) =                     {x' = x, y' = y, i' = i + 1, x ≥ a,
   nat(c1) + B(c1, x', y', a, b, i')          y ≥ b, i ≥ 1, i < 2y + x}
```

Listing 3.2. Example of a cost relation system (cost of loop 1 in method *while*).

3.2 PUBS

PUBS (*Practical Upper Bounds Solver*) is a tool to automatically obtain closed form upper bounds for cost relation systems. It is an implementation of the approach for finding closed form upper bounds in static cost analysis presented in [1], which is summarized in the Related Work section of this thesis.

In the loop cost analysis implemented as the central contribution of this thesis, for each loop in a TouchDevelop method we compose a CRS which represents the loop, and then we use PUBS to solve this system. By passing a cost relation system which captures correctly the semantics of some loop to PUBS, we will get back from PUBS a cost expression that is a upper bound for the cost of the loop.

In the remainder of this thesis, by *evaluating a CRS S* we mean evaluating the first cost relation in *S* (with respect to the cost relations in *S*). This is exactly what PUBS does if it gets as input a CRS, but no further specification about which cost relation to evaluate.

3.3 Cost of a Control Structure

In this thesis, we use program points to refer to control structures. The *program point* of a statement is a pair (r, c) of integers, where r is the row number and c is the column number of the position of this statement in the source code file. For better readability, we do not use this definition in the examples in this written report. Instead, we explicitly mark and name program points in the code listings by adding comments of the form `// program point k` (for some positive integer k). In this section, we use p and q to denote program points. By definition, at each program point in the code at most one control structure can occur, so each control structure can uniquely be identified by its program point. For simplicity, we write *control structure p* (*loop p, conditional p*) to refer to the control structure at program point p.

A cost relation captures the cost of a program – or, in the case of this thesis, more specifically the cost of a control structure – with respect to a given cost model. In this thesis we do not use a concrete way to model costs, such as a cost model which takes into account the actual cost (e.g., in terms of energy consumption) of executing a certain statement. We simply say that the cost of a block of statements is the sum of the costs of the statements in this block.

In this section we present a way to define the cost of a control structure in terms of cost parameters. A *cost parameter* is a symbolic expression that captures the cost of a set of statements. We use the following cost parameters:

- c_p is the cumulative cost of all the statements in loop p which are not a control structure themselves.
- t_p is the cumulative cost of all the statements in the *then*-branch of conditional p which are not a control structure themselves.
- f_p is the cumulative cost of all the statements in the *else*-branch of conditional p which are not a control structure themselves.

Furthermore, we need to introduce the following definitions:

- *Cost*(p) is the cost of the control structure at program point p.
- For loops, n_p is the number of times loop p is iterated. For conditionals, $n_p = 1$.

- *inner(p)* denotes a set of program points. It is defined to be *inner(p)* = {q | control structure q is a statement directly contained in control structure p}. Intuitively, this describes the inner control structures of control structure p.

Now, we can formally define the cost of a control structure:

DEFINITION: Assume that we have a control structure at program point p. Then the cost of this control structure is

$$Cost(p) = n_p\Big(c_p + \sum_{q \in inner(p)} Cost(q)\Big)$$

As an example, consider again the cost relation system in listing 3.2. When we use PUBS to solve this cost relation system, we get nat(a+2*b-1)*nat(c_1) as an upper bound of the cost of loop 1 in method *while*. We can see that this cost expression actually conforms to the definition above, where nat(a+2*b-1) corresponds to n_p, nat(c_1) corresponds to c_p, and 0 corresponds to the sum in the definition above, as loop 1 in method *while* does not have any inner control structures.

4 Sample

During the last four years, the static analyzer Sample (*Static Analyzer of Multiple Programming LanguagEs*) [6, 7] has been developed at the Chair of Programming Methodology. Sample is based on the abstract interpretation theory [4, 5]. This analyzer has been already applied to a wide range of properties and to different programming languages (Scala and Java bytecode, as well as TouchDevelop). The goal of this master thesis is to apply Sample to infer an overapproximation of the cost of loops in TouchDevelop programs relying on PUBS.

4.1 Analyses in Sample

An analysis in Sample is the composition of a heap analysis and a value analysis. Sample can be used with various heap abstractions and value domains. A state of an analysis is a pair consisting of a state of the heap domain and a state of the value domain. Sample has already been applied to various value analyses (e.g. strings [3], types [6], access permissions [8], data leaking [12]). It supports some common numerical analyses through Apron [10], which is a library dedicated to the static analysis of the numerical variables of a program. Additionally, some heap analyses have been already developed in Sample [7].

In this thesis, we run a numerical analysis (TouchDevelop analysis with Apron) on the input program. We then use the result of this analysis to overapproximate the costs of the loops in the input program. We use Apron's linear equalities (based on the work by Karr [11]) as the value domain, and we use a program point based heap abstraction.

4.2 Control Flow Graph (CFG)

Sample uses control flow graphs to represent methods. A *control flow graph (CFG)* of a method *m* is a graph representation of all paths through *m* that might be traversed when *m* is executed. In Sample, the CFG of some method *m* is represented as a directed, weighted graph. Each node of this graph is a (possibly empty) ordered list of statements (instructions and boolean expressions), representing the concatenation of statements. There is an edge from node n_1 to node n_2 if (according to the semantics of *m*) the first statement in n_2 may be executed directly after the last statement in n_1. We call an edge from node *n* to some other node an *out-edge* of *n*, and an edge from some other node to *n* an *in-edge* of *n*. If a node has one out-edge, this edge does not have a weight. If a node has two out-edges, then one of them has the label *true* as a weight and the other one has the label *false* as a weight. We will see in the next section that such nodes always are the initial point of some control structure, and the weights of the two out-edges are needed to clearly define the control flow.

4.3 From TouchDevelop Control Structures to CFGs

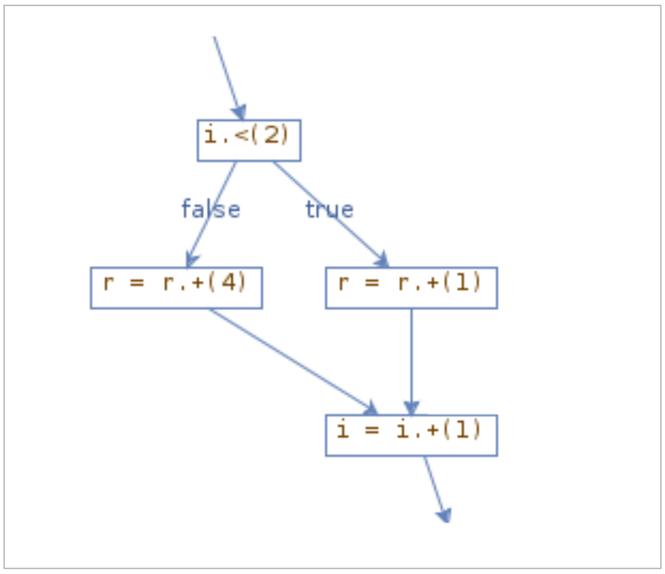

Figure 4.1. A conditional in a CFG.

We can often identify typical patterns which represent control structures in a CFG. In figure 4.1 we see how a conditional looks like in a CFG. We have a node with one in-edge and two out-edges, one labeled *true* and the other labeled *false*. We call this first node the *initial node* of the conditional. It contains exactly one statement, which is the condition of the conditional. We then have two edge-disjoint paths, each containing one or more nodes. One of them starts with the edge labeled *true*. It represents the execution of the *then*-branch of the conditional. The other path starts with the edge labeled *false*. It represents the execution of the *else-branch* of the conditional. Finally, the two paths join at a node with two in-edges and one out-edge, which we call the *final node* of the conditional. Each of the two paths of the conditional may contain inner control structures, and thus be split up in sub-paths.

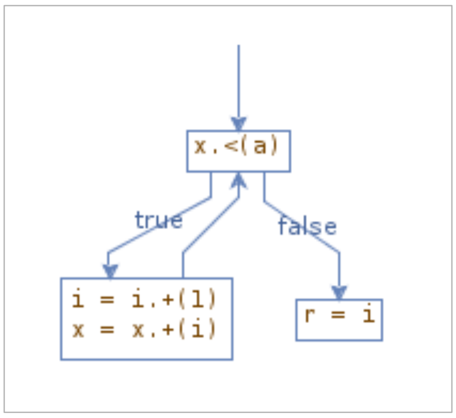

Figure 4.2. A loop in a CFG.

In a control flow graph, the information about the specific type of a loop (namely, `while`, `for`, and `foreach`) is abstracted away, since all loops are compiled to CFG structures. Consider for instance figure 4.2. We have a node with two in-edges and two out-edges, one labeled *true* and the other labeled *false*. We call this node the *initial node* of the loop. At the same time, it is the *final node* of the loop. It contains exactly one statement, which is the condition of the loop. Starting with the out-edge labeled *true*, there is a cyclic path both starting and ending at the initial node of the loop. This path, which represents one execution of the loop body, may contain inner control structures, and thus be split up in sub-paths. Basically, the structure of a loop in a CFG corresponds directly to a *while* loop as it is presented in section 2.2 of this thesis. It is easy to see how a *while* loop is converted into a loop in a CFG: The loop condition becomes the (single) statement in the initial node of the loop in the CFG, and the list of statements in the body of the while loop is converted into the cyclic path of nodes starting and ending at the initial node of the loop. For a *foreach* loop, we can see the conversion in the following way: First, the *foreach* loop is converted into an equivalent *while* loop as described by the equivalence relation given in section 2.2. Then, the resulting *while* loop is converted into a loop of the CFG as described above. The same holds, analogously, for *for* loops.

4.4 Control Flow Graph Execution (CFGE)

Given an analysis, a value domain, a heap domain, an input method *m*, and an initial state defining the context in which *m* is called, Sample runs the analysis on the CFG of *m,* which we denote by *cfg* here. The result of this process is also a directed, weighted graph *cfge* which we call a *control flow graph execution (CFGE). cfge* and *cfg* are isomorphic, i.e., they have the same structure where each node *n* in *cfg* has a corresponding node *n'* in *cfge*. If there is an edge from *n* to *m* with weight *w* (which is a label *true*, a label *false*, or no label), then there is an edge from *n'* to *m'* with weight *w* in *cfge*. Remember that each node in a CFG of Sample is an ordered list of statements. If a node *n* in *cfg* contains *x* statements, then its corresponding node *n'* in *cfge* is an ordered list of $x + 1$ states, where the *i*-th state in this list is the result that was inferred by the analysis for the program point in between the (*i*-1)-th and the *i*-th statement in *n*[1].

[1] To be precise, this only holds for $1 < i < x + 1$. The first state in *n'* is the result that was inferred by the analysis for the program point directly before the first statement in *n*, and the (x+1)-th state is the result that was inferred by the analysis for the program point directly after the *x*-th statement in *n* .

5 Approach

To overapproximate the cost of TouchDevelop loops, we use the classical approach to static cost analysis. It consists of two parts. First, in Sample we run a numerical analysis on the input program. Second, we use the result of the numerical analysis to infer a cost relation system for each loop in the input program. The first cost relation in this system represents an overapproximation for the cost of the loop. We then pass this cost relation system to PUBS for solving it. In this way, we try to get a closed-form representation of the cost defined by the cost relation, which is an upper bound for the cost of the loop we are looking at.

5.1 Step-by-Step Description

Given a method *m* as input, we perform the following steps to create a cost relation system for each loop in *m*:

1) First, we compile *m* to generate the control flow graph representing *m*.

2) We then slightly modify the result, which we call *augmenting* the control flow graph. Let's denote the resulting augmented control flow graph by *cfg*.

3) Then, we execute a numerical analysis (which uses the Apron library) on *cfg*, which gives us *cfge*.

4) Then, using *cfge*, we retrieve all the relevant information to infer the cost relation systems:
    - First, we iterate over the nodes of *cfg*. For each node, we check whether it is the initial node of a loop .
    - For each loop that we found:
        - We retrieve the loop condition, and store it in a tree-like data structure which can later be used by the loop cost analysis.
        - For each variable that appears in the loop (but not for arguments, as we assume them to be constant in the scope of the method), we try to find out how its value changes in each iteration of the loop body. In particular we try to detect if the variable value always increases, always decreases, or stays constant.

        - For each variable that appears in the loop, we try to find its initial value, i.e., its value before the loop body is executed the first time.
    - For each conditional that we found inside a loop, we retrieve its condition. We store this condition and its negation in a tree-like data structure which can later be used by the loop cost analysis.
    - We try to find out about the topology of *cfg*, i.e., which control structure is contained in which other control structure.

5) For each loop in *m*, we now try to compute an upper bound of its cost. We do this by performing the following steps:
    - We compose a cost relation system that represents the loop, using the information we gathered at step 4).
    - We pass this cost relation system to PUBS. We use the answer that we get back from PUBS as the result of our analysis of this loop.

5.2 Implementation

In the scope of this thesis, two additions to Sample were implemented:

- The *TouchDevelop cost analysis compiler* (chapter 6), which augments the control flow graph produced by the TouchDevelop compiler to infer useful cost relations.
- The *TouchDevelop loop cost analysis* (chapter 7), which must be used in combination with the *TouchDevelop cost analysis compiler* (else, the analysis produces an error). It does the same as the existing *TouchDevelop analysis with APRON*, and additionally tries to find an upper bound for the cost of all loops in the methods under examination.

Both the analysis and the compiler can be selected in the GUI of Sample. When running the *TouchDevelop loop cost analysis*, the *Loop Cost* or the *Loop Cost with Show Graph* property must be selected such that the loop costs are actually calculated and displayed.

6 TouchDevelop Cost Analysis Compiler

The loop cost analysis uses a slightly modified version of Sample's TouchDevelop compiler. It mainly performs a pre-processing step needed by the loop cost analysis, which we call *augmenting* a control flow graph.

6.1 Augmenting a Control Flow Graph

The loop cost analysis does not operate on a control flow graph as it is normally created by Sample, but on a slightly modified version of such a graph, which we call an *augmented control flow graph*. Creating an Augmented control flow graph out of a control flow graph *cfg* (which we call augmenting *cfg*) means performing the following steps for each loop in *cfg*:

1) Find all variables that appear in the loop. We denote the set of these variables by V.
2) For each variable $v \in V$, add a new assignment *old_v* := v at the beginning of the loop, directly after the loop condition and before any other existing statement of the loop body. This adds a variable *old_v* to *cfg*, which does not exist in the original control flow graph. We call such a variable a *old-variable*. As *old_v* is not affected by any other statement in the loop, after an execution of the loop body its value equals v's value before this particular execution of the loop body.

This last point gives us a means to argue about how the value of each variable v changes when the body of a loop is executed once. Because the loop cost analysis needs to know entry-exit relations on the variables involved in the loop, we actually use an augmented CFG rather than an unmodified CFG as input for the analysis.

```
action sumOfSums(a: Number) returns r: Number {
    $x := 1;
    $i := 1;
    ;                           // program point 1
    while ($x < $a) do {        // program point 2
        $i := $i + 1;
        $x := $x + $i;
        ;                       // program point 3
    }
    $r := $i;
```

Listing 6.1. Code of method *sumOfSums*.

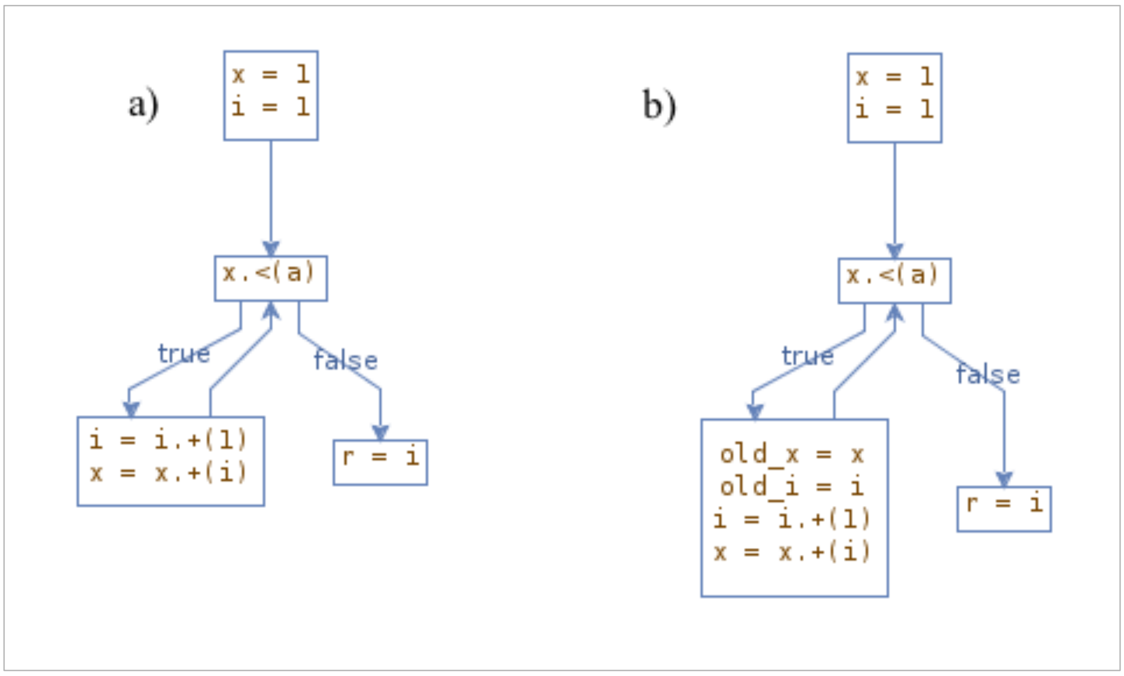

Figure 6.1. a) the CFG of method *sumOfSums;* **b)** the augmented CFG of method *sumOfSums*.

As an example, consider method *sumOfSums* (Listing 6.1; we will discuss the result of analyzing this method in section 8.10). In figure 6.1 a) we see the CFG that Sample produces when it compiles *sumOfSums* using its TouchDevelop compiler. It contains a simple loop with the loop condition `x < a`, where `a` is an argument of the method. When we augment it, we see that we have a single loop at program point 1, so we have to perform steps 1) and 2) for this loop. We first determine the set of all variables appearing in the loop, which turns out to be $V = \{i, x\}$. For each variable in V, we add an assignment to the loop, which gives us the assignments `old_i = i` and `old_x := x`. Note that we do not get an assignment `old_r := r`, as r is a variable of the method, but does not appear inside the loop. According to the description above, we add the two new statements directly after the loop condition and before any other existing statement in the loop body, i.e., directly before `i := i + 1`. The resulting augmented CFG is shown in figure 6.1 b). Note that our implementation always adds the new assignments to the node containing the first statement of the loop body, but they could be added to the node with the loop condition as well, or even to a newly created node. This does not matter, as long as the new assignments are placed at the correct position relative to the existing statements.

7 TouchDevelop Loop Cost Analysis

In this chapter we present the implementation of the actual analysis. In particular, we explain how it uses the result of a numerical analysis to infer a cost equation system which represents the cost of a loop. Let us assume that we have a method *m* as the input for our analysis. *cfg* denotes the augmented CFG representing *m*, and the control flow graph execution *cfge* is the result of running a numerical analysis (using Apron's linear equalities) on *cfg*.

7.1 Finding the Control Structures

To find all control structures, we iterate over the nodes of *cfg*. For each node, we check whether it is the initial node of a loop or a conditional. Remember that the *initial node* of a control structure is the node that contains the condition of the loop or conditional, respectively.

We use the program point where the control structure begins (syntactically) in the source code as an identifier for that control structure. As no two control structures can start at the same program point, these identifiers are guaranteed to be unique.

As soon as the analysis knows which control structures there are in *cfg*, we can also find out about the topology of the input method, i.e., which control structure is contained in which other control structure. For control structures in conditionals, we additionally distinguish between being contained in the *if*-branch and being contained in the *else*-branch.

Both for finding the control structures and finding out about their nesting, our implementation performs a breadth-first search [13] on the control flow graph *cfg*.

7.2 Loop Condition

Our loop cost analysis only works if the loop under examination has a loop condition which fulfills certain properties. We call such a loop condition a valid condition.

DEFINITION: A valid condition is one of:

- `true` (represented as the linear relation `1 = 1`)
- `false` (represented as the linear relation `0 = 1`)
- a linear relation *expr1 op expr2*, where *expr1* and *expr2* are linear arithmetic expressions, and *op* is a relational operator, $op \in \{=, \neq, <, \leq, >, \geq\}$
- $\neg c$, the negation of a valid condition *c*
- $c1 \wedge c2$, the conjunction of two valid conditions *c1* and *c2*
- $c1 \vee c2$, the disjunction of two valid conditions *c1* and *c2*

Note that this recursive definition allows us to have nested loop conditions, such as $(i > 1 \wedge j > 10) \vee \neg (k < i)$. We call a condition which consists of one or more conjunctions, and/or disjunctions, a *composite condition*. PUBS does not allow to have a composite as a guard of a cost equation (only linear relations are allowed). However, we found a way to still be able to handle them, namely by splitting them up in multiple cost relations (see section 7.6).

The loop cost analysis retrieves the loop condition from the semantics of the method we are looking at. If it is not a valid condition, the analysis can not compute the cost of the loop, because in this case the cost of the loop can not be represented as a cost relation system that can be handled by PUBS. If it is a valid condition, the loop condition is stored in a tree-like data structure. Each operation of the condition is a node of the tree, where the outermost operation is the root of the tree. Conjunctions and disjunctions are intermediate nodes of the tree, while the basic linear relations are the leaves of the tree.

Note that no node of the tree represents a negation. There is no need for this, as every valid condition can be converted to an equivalent boolean expression with no explicit negation in the following way (and the analysis exactly does this): First, we repeatedly apply the laws of DeMorgan $\neg (c1 \wedge c2) \equiv (\neg c1) \vee (\neg c2)$ and $\neg (c1 \vee c2) \equiv (\neg c1) \wedge (\neg c2)$ on the conjunctions and disjunctions, respectively, starting with the outermost ones and proceeding inward. This gives us an equivalent boolean expression which only contains negations of basic linear relations, but no negations of conjunctions or disjunctions. Then, we apply the arithmetic rules $\neg (a = b) \equiv (a \neq b)$, $\neg (a \neq b) \equiv (a = b)$, $\neg (a < b) \equiv (a \geq b)$, $\neg (a \leq b) \equiv (a > b)$, $\neg (a > b) \equiv (a \leq b)$ and $\neg (a \geq b) \equiv (a < b)$ to get rid of the explicit negations. The absence of explicit negations in our representation of the loop condition is helpful later on when we compose the cost relation system.

As an example, consider again method *sumOfSums* which we introduced in chapter 6. Here, we have $x < a$ as the loop condition, which is the only statement in the initial node of the CFG (figure 6.1 b)). This is clearly a valid condition, as it is a linear relation.

7.3 Variable Update Rules

We call how the value of a variable *v* changes in one iteration of a loop the *variable update rule* for *v*. For example, the update rule of the iteration variable *i* in a TouchDevelop *for* loop is $i' = i + 1$, as *i* is increased by 1 in each execution of the loop body. Here, *i* denotes the value of the iteration variable before the execution of the loop body, and *i'* refers to the value of the iteration variable after the execution of the loop body.

The *loop variables* of a loop *l,* written as *LV(l)* is the set of the variables that have an influence on the number of times *l* is iterated. In the step discussed here, the loop cost analysis iteratively finds *LV(l)* by performing the following algorithm:

1) Add all variables that appear in the loop condition to an initially empty set *S*.
2) Try to find an update rule for every variable in *S* for which we have not tried this before.
3) Add all variables that appear in the update rules found in the previous step to *S*. If, by doing this, *S* grows (i.e., we found at least one new variable), repeat steps 2) and 3).
4) *LV(l) := S*

Our analysis can only find variable update rules that have a certain form (see below). Remember that the loop cost analysis uses the control flow graph execution *cfge* as input, i.e., the result of executing a numerical analysis on *cfg*. In order to find the variable update rules, our analysis uses this result. It looks at all linear relations that the numerical analysis infers at the last statement in the last node of the loop. Now, the *old-variables* which were added when the CFG was augmented become helpful. Thanks to them, the numerical analysis is potentially able to infer relations between the value of a variable *v* after the loop body was executed and its value before the loop body was executed (represented by *old_v),* and our analysis gets these relations as part of the set of linear constraints it gets from the numerical analysis.

To find the update rule for a variable *i*, the loop cost analysis performs an algorithm consisting of two runs, where the second run is only performed if the first one was not successful[2]:

1) In the first run, the analysis considers only the linear constraints coming from the numerical analysis that are fully described by *v* and *old_v,* i.e., that have a coefficient of zero for all other variables. It looks for a constraint of the form $v = a*old\_v + b$, where a and b are rational constants, with $a \neq 0$. In case of success, it translates the found constraint to the update rule $v' = a*v + b$.

2) In the second run, the analysis considers all linear constraints coming from the numerical analysis. It looks for a constraint of the form $v = expr$, where exp*r* is a linear expression. In case of success, it translates the found constraint to the update rule $v' = expr*$, where *expr** is the result of replacing all occurrences of *old_v* by *v* in *expr*.

As an example, consider again method *sumOfSums*. The program point after the last statement of the loop body is program point 3. For this program point, we get the following constraint from the numerical analysis:

$$i = old\_i + 1$$
$$x = old\_x + i$$

By performing the algorithm described above, the analysis finds variable update rules for both loop variables. From $i = old\_i + 1$ it derives the update rule $i' = i + 1$ in the first run of the algorithm. From $x = old\_x + i$ it derives the update rule $x' = x + i$, but only in the second run of the algorithm, as the updated value of *x* depends on the value of some other loop variable, namely *i*.

Using the detected variable update rules, the loop cost analysis then also tries to find out for each loop variable if its value increases or decreases, according to the following definitions[3]:

[2] Note that in Sample *v* refers to the new variable value (and *old_v* refers to the old variable value), while in the update rule (which is used as an input for PUBS) *v* refers to the old variable value (and *v'* refers to the new variable value). This is just a notational matter due to implementation reasons, and does not cause any problems.

[3] We are aware that according to these definitions it may become impossible to determine whether two or more variables increase (or decrease, respectively) due to cyclic dependencies in their update rules. To prevent non-termination, our implementation tries to detect increasing and decreasing variables in a loop. As soon as in a loop iteration no further variable could be detected to be increasing or decreasing, the loop terminates.

- The value of a variable *v* increases if it has an update rule of the form *v'* = *expr*, with *expr* = $\left(\sum_{i=1}^{n} coeff_i * v_i \right) + c$, where the v_i are variables other than *v*, the $coeff_i$ are their coefficients (which must be rational numbers), and *c* is a constant (which must be rational as well), such that the values of all variable-coefficient-pairs in *expr* increase and *c* is non-negative.
- Analogously, the value of a variable *v* decreases if it has an update rule of the form *v'* = *expr*, with *expr* = $\left(\sum_{i=1}^{n} coeff_i * v_i \right) + c$, where the v_i are variables other than *v*, the $coeff_i$ are their coefficients (which must be rational numbers), and *c* is a constant (which must be rational as well), such that the values of all variable-coefficient-pairs in *expr* decrease and *c* is non-positive.
- The value of a variable-coefficient-pair $coeff_i$*v_i increases if the value of v_i increases and $coeff_i$ is non-negative, or if the value of v_i decreases and $coeff_i$ is non-positive. Analogously, the value of a variable-coefficient-pair $coeff_i$*v_i decreases if the value of v_i decreases and $coeff_i$ is non-negative, or if the value of v_i increases and $coeff_i$ is non-positive.

Note that our definition of an increasing or decreasing variable value also includes the case where the variable value is constant. When we know that in a loop the value of a variable *v* increases (or decreases, respectively), we can – together with our knowledge about the initial value of v – derive loop invariants of the form v ≤ *expr* or v ≥ *expr* or v = *expr*, respectively, which can then be added as additional guards to the cost relation system. This is described in the next section.

For our example method *sumOfSums*, the analysis first detects that the value of *i* increases. Using this, it then detects that the value of *x* increases as well (because the value of *i* increases).

7.4 Initial Variable Values

For each loop variable, the loop cost analysis tries to find the initial value, i.e., its value before the body of the loop is executed for the first time. This works like the detection of the variable update rules described in the previous section, i.e., the analysis uses the result from the numerical analysis here as well. It looks at all linear constraints that the numerical analysis infers at the program point just before the loop, i.e., after the last statement in the node before the loop.

When trying to find the initial value of a variable v, the analysis looks for a linear constraint of the form $v = expr$, where $expr$ is a linear expression. In case of success, the information about the initial value of v is stored in memory, and will be used when the cost relation system is created (section 7.6).

When the analysis could find the initial value of a variable v with respect to a loop l, and if it could furthermore detect that in l the value of v increases, decreases, or stays constant (section 7.3), then it can derive a loop invariant of the form $v \leq v_{init}$ or $v \geq v_{init}$ or $v = v_{init}$, respectively, where v_{init} denotes the initial value of v. This linear relation can then be added as an additional guard to the cost relation system that represents the cost of l (see section 7.6).

As an example, consider again method *sumOfSums*. The program point just before the loop is program point 1. For this program point, we get the following constraint from the numerical analysis:

$$i = 1$$
$$x = 1$$

This directly gives us the initial values for both loop variables. Combining this with the information about increasing/decreasing variable values from the previous section (remember that we found out that both loop variables increase), the analysis is able to infer the linear relations $i \geq 1$ and $x \geq 1$ which can be added as additional guards to the cost relation system.

7.5 Conditional Condition

In section 7.2 we saw that the loop cost analysis can handle loops which have a valid condition as their condition. The same holds for conditionals. A conditional's condition is retrieved in exactly the same way (using the result of the numerical analysis) as we saw it for loop conditions in section 7.2.

As we will also have to include the semantics of the *else*-branch in the cost relation system, we additionally store a tree-like structure representing the negation of the condition in memory at this point. This is not absolutely necessary, as we can always retrieve the negation of a boolean expression from the expression itself. However, it will be convenient to have the negated condition available in this form later on when we compose the cost relation system (section 7.6).

7.6 Composing the Cost Relation System

At this point the loop cost analysis has gathered all the information needed to create a cost relation system, whose first equation represents the cost of the loop under examination. The analysis can then pass this system to PUBS. We can be sure that this cost relation system correctly overapproximates the cost of the loop under examination, because all the constraints that we use to generate the cost relation system are inferred by a sound abstract numerical domain. Therefore, they are a sound overapproximation of the concrete behavior of the input program. We use four examples to illustrate how the resulting cost relation system looks like. First, we look at the base case of a loop where we have no inner control structures, and no composite condition (i.e., the condition is simply a linear relation here). Then, we analogously consider the base case for a conditional. Next, we explain how the cost relation system must be adapted when we have inner control structures. Finally, we show how the cost relation system is composed if the loop or conditional, respectively, has a composite condition.

Base case (loop)

Here, we use method *while* (listing 7.1) to illustrate how the loop cost analysis composes the corresponding cost relation system. The method contains a simple *while* loop which is iterated as long as the counter variable *i,* which is initially 1, is smaller than *x + 2y*, which has a constant value defined by two arguments of the method, *a* and *b*.

```
action while(a: Number, b: Number) {
    $x := $a;
    $y := $b;
    $i := 1;
    while $i < $x + 2 * $y do {       // program point 1
        $i := $i + 1;
    }
}
```

1) $A(c_1, a, b) = B(c_1, a, b, a, b, 1)$ $\{true\}$

2) $B(c_1, x, y, a, b, i) = nat(c_1) + B(c_1, x', y', a, b, i')$ $\{x' = x, y' = y, i' = i + 1, x \geq a, y \geq b, i \geq 1, i < 2y + x\}$

Listing 7.1. Code of method *while* and CRS of the loop at program point 1.

The cost relation system composed by the loop cost analysis basically consists of two cost equations. Equation 2) represents the cost of the loop in terms of all loop variables, method arguments, and parameters of the cost model. But we do not want to get as a result a cost expression in terms of variables which appear only locally inside the method. This is why we add equation 1) to the system as the first cost equation. It gives us the cost of the loop in terms of the method arguments and cost model parameters only. This is exactly what we want, and because the answer we get from PUBS is always the cost defined by the first equation in the cost relation system, we list it as the first equation. As we can see in the example, equation 1) refers to equation 2), in which all loop variables are instantiated by the initial values of the corresponding variables, which we have retrieved before.

Equation 2) is the central part of the cost relation system. An evaluation of it represents the effects of one execution of the loop body. In our example, the first evaluation of B (which is its outermost evaluation, as the equation contains itself recursively) corresponds to executing the loop in a state with `x = a`, `y = b` and `i = 1`. The next evaluation of B then corresponds to executing the loop in a state with `x = a`, `y = b` and `i = 2`, and so on. The cost expression in equation 2) consists of two parts. The first part, which is $nat(c_1)$ in our example, is the cost of the current iteration of the loop. The second part, which is $B(c_1, x', y', a, b, i')$ in our example, is the cost of all remaining iterations of the loop after the current one. As we can see in the example, this second part occurs as a recursive evaluation of B in the cost relation system, in which each loop variable *v* is replaced by a variable *v'*. Here, *v* represents the value of the variable at the beginning of the current iteration of the loop, and *v'* represents the value of the variable at the beginning of the next iteration of the loop. This change of the variable value between two consecutive iterations of the loop is exactly what we already know as the update rule of a variable. Our analysis adds the variable update rules as guards to equation 2), and thus ensures that the primed variables (*x'*, *y'* and *i'* in the example) are clearly and correctly defined whenever B is evaluated. But the cost relation system is not complete at this point. We also need to add the loop condition, which we also retrieved earlier on, to the cost relation system. As said before, in this section we only look at the case where the loop condition is a linear relation, e.g. `i < x + 2*y` in our example. As long as this holds, we can simply add the loop condition as an additional guard to cost equation 2). This then completes the cost relation system.

Base case (conditional)

In this section we use method *if* (listing 7.2) to illustrate how the loop cost analysis composes a cost relation system from a conditional. The method simply contains a conditional. The *then*-branch of the conditional is executed if *a*, which is an argument of the method, is non-negative. As a consequence, the *else*-branch is executed if *a* is negative.

```
action if(a: Number) returns r: Number {
    if ($a >= 0) then {       // program point 1
        $r := 1;
    } else {
        $r := -1;
    }
}
```

1) $A(t_1, f_1, a) = B(t_1, f_1, a) + C(t_1, f_1, a)$ $\{true\}$

2) $B(t_1, f_1, a) = nat(t_1)$ $\{a \geq 0\}$

3) $C(t_1, f_1, a) = nat(f_1)$ $\{a < 0\}$

Listing 7.2. Code of method *if* and CRS of the conditional at program point 1.

In the case of conditionals, the cost relation system composed by the loop cost analysis basically consists of three cost relations (more if we have a composite loop condition, or if we have inner data structures). We get the cost of the conditional by evaluating equation 1). As we can see in the example, this cost is simply defined as the sum of the costs defined by equation 2) and equation 3), respectively. Equation 2) represents the cost of executing the *then*-branch of the conditional, while equation 3) represents the cost of executing the *else*-branch. The cost expression that we get by evaluating equation 2) is simply an expression in terms of the parameters of the cost model we use. As executing a conditional involves no repetitions, there is no recursive evaluation of B in the cost expression, as we had it before for the loop case. The same holds analogously for equation 3). In order to complete the cost relation system, all we need to do is to add the conditional's condition as a guard to equation 2), and its negation as a guard to equation 3).

Nested control structures

Here, we use method *nested* (listing 7.3) to illustrate how the cost analysis composes the CRS when we have a control structure which contains another control structure. The method contains two nested *for* loops. The inner loop is iterated as long as the counter variable *j,* which is initially 0, is smaller than *b*, which is an argument of the method. The outer loop is iterated as long as the counter variable *i,* which is initially 0, is smaller than *a*, which is also an argument of the method.

```
action nested(a: Number, b: Number) returns r: Number {
    $r := 0;
    for 0 ≤ i < $a do {            // program point 1
        for 0 ≤ j < $b do {        // program point 2
            $r := $r + 1;
        }
    }
}
```

1) $A(c_1, c_2, a, b) = B(c_1, c_2, 0, b, a, a, b)$	$\{true\}$
2) $B(c_1, c_2, j, j\_bound, a, i\_bound, i, b)$ $= nat(c_1) + B(c_1, c_2, j', j\_bound', a, i\_bound', i', b) + C(c_2, b)$	$\{i' = i + 1, i\_bound' = i\_bound, i \geq 0, i\_bound = a, i < i\_bound\}$
3) $C(c_2, b) = D(c_5, 0, b, b)$	$\{true\}$
4) $D(c_2, j, j\_bound, i, b)$ $= nat(c_2) + M_1(c_2, j', j\_bound', b)$	$\{j' = j + 1, j\_bound' = j\_bound, j \geq 0, j\_bound = b, j < j\_bound\}$

Listing 7.3. Code of method *nested* and CRS of the loop at program point 1 (the "outer" loop). Note that the CRS of the loop at program point 2 (the "inner" loop) would simply consist of cost relations 3 and 4.

Including information about nested control structures in the cost relation system is simple and straightforward. We can compose a cost relation system for the cost of the inner control structure (e.g., the inner loop in the example here) as described above. Then, we simply add the cost relations of this system to the cost relation system for the outer control structure. In the example, equations 3) and 4) in the cost relation system for the outer loop are actually the two cost relations defining the cost of the inner loop. We know that at each iteration of the outer loop, the inner control structure is executed once. So, all we need to do is adding the cost of the inner control structure (which is

`C(c₂, b)` in our example) to the cost expression in equation 2). If the outer control structure is a conditional, we do it analogously.

Composite conditions

When we have a loop with a condition that contains one or more conjunctions and/or disjunctions, which we call a composite condition, the loop cost analysis must ensure that this composite condition is correctly represented in the cost relation system it generates. Remember that in our implementation a condition is stored as a tree-like structure, where each operation of the condition is a node of the tree, and the outermost operation is the root of the tree. Conjunctions and disjunctions are intermediate nodes of the tree, while the basic linear relations are the leaves of the tree. When composing the cost relation system, our analysis creates up to two cost relations for each node in this tree.

```
action compositeCondition() returns r: Number {
    $i := 0;
    $j := 0;
    while ($i < 3) or ($j ≥ 5 and $j < 20) do {   // program point 1
        $i := $i + 1;
        $j := $j + 2;
        $r := $r + $i + $j;
    }
}
```

1) $A(c_1) = B(c_1, 0, 0)$	{true}
2) $B(c_1, i, j) = C(c_1, i, j)$	{true}
3) $B(c_1, i, j) = D(c_1, i, j)$	{true}
4) $C(c_1, i, j) = nat(c_1) + B(c_1, i', j')$	$\{i' = i + 1, j' = j + 2, i \geq 0, j \geq 0, i < 3\}$
5) $D(c_1, i, j) = E(c_1, i, j)$	{true}
6) $E(c_1, i, j) = F(c_1, i, j)$	$\{j \geq 5\}$
7) $F(c_1, i, j) = nat(c_1) + B(c_1, i', j')$	$\{i' = i + 1, j' = j + 2, i \geq 0, j \geq 0, j < 20\}$

Listing 7.4. Code of method *compositeCondition* and CRS of the loop at program point 1.

We use method *compositeCondition* (listing 7.4) to illustrate how this is exactly done. The loop in *compositeCondition* has a condition which is a disjunction, where the left operand is a linear relation and the right operand is the conjunction of two linear relations.

- Equation 1) is exactly the same as before: It is used to specify the initial values of the loop variables. The outermost operation in our condition is the disjunction.
- Equations 2) and 3) represent this disjunction. Notice that they both define B. In this way, they capture the semantics of the disjunction: The disjunction evaluates to true if either of its operands evaluates to true. Analogously, when calculating the cost of the loop that has this disjunction as its condition, PUBS can use either of the two definitions for it, each representing one of the operands.
- Equation 4) represents the left operand of the disjunction, which is the linear relation `$i < 3`. As this equation represents a basic linear relation, it has the same form as Equation 2) in the base case. It has a cost expression consisting of two parts (the cost of the current loop iteration, and the sum of the cost of all future loop iterations), as we have seen it before. As in equation 2) of the base case, the guards of the equation are the variable update rules, and the information about the lower bounds of *i* and *j*. Additionally, we add the part of the condition that we are currently looking at, namely the linear relation `$i < 3`, as a guard.
- Equation 5) represents the conjunction that we have as an inner operation in our example condition. Notice that, compared to the disjunction case discussed before, we do not have two definitions of D. Instead, we have some kind of sequential evaluation of cost relations here: D is defined by E, which in turn is defined by F, so in order for D to have a non-zero value, all the guards in equations 6) and 7) must evaluate to true. This fully reflects the semantics of the conjunction: A conjunction evaluates to true exactly if all its operands evaluate to true.
- Equations 6) and 7) represent the operands of the conjunction, namely the linear relations `j ≥ 5` and `j < 20`. Equation 6) simply checks the guard `j ≥ 5`. Finally, Equation 7) has the same form as Equation 4) in this example (or Equation 2 in the base case, respectively). The guards of the equation are the variable update rules, and the information about the lower bounds of *i* and *j*. Additionally, we use the part of the condition that we are currently looking at, namely `$j < 20`, as a guard.

8 Case Studies (Constructed Scripts)

In this chapter, we present the result of the loop cost analysis for some input scripts we wrote ourselves, and which we used as test cases during the development of our implementation. The results nicely illustrate the capabilities and the limits of the loop cost analysis.

8.1 Simple *for* Loop

Method *sum* (listing 8.1) calculates the sum of the numbers 0 to *a*-1, where *a* is given as an argument of the method. To do so, it uses a simple *for* loop. The return value *r* does not appear in the loop condition, nor does it influence any variables which do (namely *i*), so it does not have an influence on the cost of the loop. It is included in the example only for illustrative purposes.
The loop condition is simply `i < a`, which is obviously a linear relation. As *a* is an argument (having a non-modifiable value) and the update rules `i' = i + 1` and `i_bound' = i_bound` (which are linear as well) can be found by the loop cost analysis, the analysis is able to find nat(*a*)*nat(c_1) as an upper bound for the loop cost, which is correct (and, in this simple case, also precise).

Note that equation 2) of the cost relation system must contain some information about the lower bound of *i* (namely the guard `i ≥ 0`). Without it, PUBS would not be able to find a ranking function for the system, and thus to calculate an upper bound for the loop cost.

```
action sum(a: Number) returns r: Number {
    $r := 0;
    for 0 ≤ i < $a do {        // program point 1
        $r := $r + $i;
    }
}
```

1) $A(c_1, a) = B(c_1, 0, a, a)$ $\{true\}$

2) $B(c_1, i, i\_bound, a)$ $\{i' = i + 1, i\_bound' = i\_bound,$
$= nat(c_1) + B(c_1, i', i\_bound', a)$ $i \geq 0, i\_bound = a, i < i\_bound\}$

Result: nat(*a*)*nat(c_1)

Listing 8.1. Code of method *sum*, CRS of the loop at program point 1, and result of the analysis.

8.2 *foreach* Loop over Immutable Collection

foreachPic (listing 8.2) is a simple method that displays all pictures that can be found on the phone on which the method runs. To do so, it uses a foreach loop that iterates over the immutable collection *media→pictures*.

Here, we can take advantage of the fact that Sample treats *foreach* loops in exactly the same way as *for* loops. In the CFG, the loop condition is rewritten as

```
while pic_index < pic_collection.count().
```

From Sample we get that *pic_collection.count()* is equal to *Length({9,3})* (i.e., the length of the data structure at program point 9,3), which in turn is equal to *Length(media.pictures)*. Using all this, the loop cost analysis is able to compose the cost relation system, which has the same structure as the one derived from the simple *for* loop from section 8.1: we have *pic_index* instead of *i*, *Length({9,3})* instead of *i_bound*, and *Length(media.pictures)* instead of *a*. As in section 8.1, the loop condition `i < i_bound` is a linear relation, and we have linear update rules for *pic_index* and *Length({9,3})*, so the analysis is able to find a correct and precise upper bound of the cost of the *foreach* loop, namely nat(*Length(media.pictures)*)*nat(c_1).

```
action foreachPic() {
    foreach p in media→pictures do {      // program point 1
        $p→post_to_wall;
    }
}
```

1) A(c_1, Length(media.pictures)) = B(c_1,0,Length(media.pictures), Length(media.pictures))	{true}
2) B(c_1, p_index, Length({9,3}), Length(media.pictures)) = nat(c_1) + B(c_1, p_index′, Length({9,3})′, Length(media.pictures))	{p_index′ = p_index + 1, p_index ≥ 0, Length({9,3})′=Length(media.pictures), Length({9,3}) = Length(media.pictures), p_index < Length({9,3})}

Result: nat(*Length(media.pictures)*)*nat(c_1)

Listing 8.2. Code of method *foreachPic*, CRS of the loop at program point 1, and result of the analysis.

8.3 *foreach* Loop over Mutable Collection

Method *foreachNum* (listing 8.3) creates a new number collection, adds two numbers to it, and then, using a *foreach* loop, prints on the phone's screen all the numbers that are initially in the collection. Additionally, in each loop iteration, another number is added to the collection.

This example is basically handled in the same way as example 8.2, where we iterated over a predefined, immutable collection. As in example 8.2, the cost relation system has the same structure as the one for the simple *for* loop (example 8.1). We have the loop condition `num_index < Length({18,3})`, which is a linear relation, and the variable update rule `num_index' = num_index + 1`, which is linear as well. So the analysis is able to find 2*nat(c_1) as an upper bound for the loop cost, which is correct and precise.

Note that adding another number to *numbers* in the loop body has no influence on the number of times that the *foreach* loop is iterated. This is due to the semantics of the TouchDevelop *foreach* loop: The loop actually iterates over a copy of `$numbers` (which is made before the first execution of the loop body), while `$numbers` still refers to the original collection, so `$numbers→add(3)` adds the number 3 to the original number collection, and not to the copy of it over which we iterate.

```
action foreachNum() {
    $numbers := collections→create_number_collection;
    $numbers→add(1);
    $numbers→add(2);
    foreach n in $numbers do {       // program point 1
        $numbers→add(3);
        $n→post_to_wall;
    }
}
```

1) $A(c_1) = B(c_1, 0, 2)$	{true}
2) $B(c_1$, n_index, Length({18,3})) = nat(c_1) + $B(c_1$, n_index', Length({18,3}))	{n_index' = n_index + 1, n_index ≥ 0, Length({18,3}) = 2, n_index < Length({18,3})}

Result: 2*nat(c_1)

Listing 8.3. Code of method *foreachNum*, CRS of the loop at program point 1, and result of the analysis.

8.4 Simple *while* Loop

In this example (method *while*, see listing 8.4) we have a *while* loop, which iterates from 1 to *x*-2*y*-1, where the values of *x* and *y* are defined by the arguments *a* and *b*, respectively. Here we can again take advantage of the fact that Sample does not make a difference between *for*, *foreach* and *while* loops when creating the CFG. Because the loop cost analysis operates on a CFG, this example can be handled in exactly the same way as example 8.1, where we had a simple *for* loop.

Here, the loop condition is `i < 2y + x`, which is a linear relation. Furthermore, we have the variable update rules `i' = i + 1`, `x' = x` and `y'= y`, which are linear as well. So the analysis is able to find nat(*a*+2**b*-1)*nat(c_1) as an upper bound for the loop cost. This is a correct and precise upper bound.

```
action while(a: Number, b: Number) {
    $x := $a;
    $y := $b;
    $i := 1;
    while $i < $x + 2 * $y do {        // program point 1
        $i := $i + 1;
    }
}
```

1) $A(c_1, a, b) = B(c_1, a, b, a, b, 1)$	$\{true\}$
2) $B(c_1, x, y, a, b, i) = nat(c_1) + B(c_1, x', y', a, b, i')$	$\{x' = x, y' = y, i' = i + 1, x \geq a, y \geq b, i \geq 1, i < 2y + x\}$

Result: nat(*a*+2**b*-1)*nat(c_1)

Listing 8.4. Code of method *while*, CRS of the loop at program point 1, and result of the analysis.

8.5 *while* Loop with Decreasing Counter Value

So far, we only looked at examples where the value of the counter variable increases. Of course, the loop cost analysis is not limited to such cases. In method *decreasing* (listing 8.5) we have a variable *i* with an initial value *upper*, which is given as an argument of the method. A *while* loop is then executed as long as *i* is equal or larger than the constant *lower*, which is also an argument of the method. The value of *i* is decreased by 1 in each loop iteration.

By looking at the cost relation system created by the loop cost analysis, we see that this method can be handled in exactly the same way as some *while* or *for* loop where the value of the iteration variable is increased linearly, such as the simple *for* loop in example 8.1. For PUBS it does not make a difference whether the variable update rule for *i* is `i' = i – 1` or `i' = i + 1`. Both can be handled (since both are linear equations). However, note that in equation 2) of the cost relation system we now have to provide some information about the *upper* bound of *i* (namely the guard `i < upper`). Only when this information is available, PUBS can find a ranking function of the system, and calculate an upper bound for the loop cost.

```
action decreasing(upper: Number, lower: Number) returns r: Number {
    $i := $upper;
    while $i >= $lower do {       // program point 1
        $i := $i - 1;
        $r := $r + 1;
    }
}
```

1) $A(c_1, lower, upper)$ $\{true\}$
   $= B(c_1, upper, lower, upper)$

2) $B(c_1, i, lower, upper)$ $\{i' = i - 1, i < upper, i \geq lower\}$
   $= nat(c_1) + B(c_1, i', lower, upper)$

Result: nat(*upper*-*lower*+1)*nat(c_1)

Listing 8.5. Code of method *decreasing*, CRS of the loop at program point 1, and result of the analysis.

8.6 *while* Loop with Exponentially Growing Counter Value

So far, we only looked at examples where the values of the loop variables are increased or decreased linearly. As long as we have such cases (where the update rules for all involved variables can be explained in terms of linear expressions), we are almost guaranteed to get a correct and precise upper bound for the loop cost, as PUBS works well with linear constraints.

Method *logarithmic* (listing 8.6) executes a *while* loop as long as the iteration variable i (which has initially the value 1) is smaller than some constant x. The value of i is doubled in each iteration of the loop, so it grows exponentially and we can expect the cost of the loop to be some logarithmic expression. The cost relation system composed by the loop cost analysis has the well-known structure: The loop condition is `i < x`, which is a linear relation, and the variable update rules `x' = x` and `i' = 2i` are linear as well. It seems that PUBS is able to find a good ranking function for the system, as we get log(2,1+nat(a-1))*nat(c_1) as an upper bound for the loop cost, which is a logarithmic expression as expected. It is a correct and precise bound.

```
action logarithmic(a: Number) returns r: Number {
    $x := $a;
    $i := 1;
    while $i < $x do {       // program point 1
        $i := $i * 2;
        $r := $r + 1;
    }
}
```

1) $A(c_1, a) = B(c_1, 1, a, a)$	$\{true\}$
2) $B(c_1, i, x, a)$ $= nat(c_1) + B(c_1, i', x', a)$	$\{i' = 2*i, x' = x, i \geq 1, x \geq a, i < x\}$

Result: log(2,1+nat(a-1))*nat(c_1)

Listing 8.6. Code of method *logarithmic*, CRS of the loop at program point 1, and result of the analysis.

However, as soon as we make a slight modification to our input method, we do not get such a nice result any more. Consider method *logarithmic2* (listing 8.7), which is identical to *logarithmic* up to one additional instruction in the loop body, `$x := $x - 1`, such that x is not a constant any more. By thinking about the semantics of *logarithmic* and *logarithmic2*, we see that any upper bound for the loop in method *logarithmic* must also be an upper bound for the cost of the loop in method *logarithmic2*: Because we only decrease the value of x, for the same value of a the loop in *logarithmic2* is executed mostly as many times as the one in *logarithmic* before x is reached.

```
action logarithmic2(a: Number) returns r: Number {
    $x := $a;
    $i := 1;
    while $i < $x do {       // program point 1
        $i := $i * 2;
        $x := $x - 1;
        $r := $r + 1;
    }
}
```

1) $A(c_1, a) = B(c_1, 1, a, a)$	$\{true\}$
2) $B(c_1, i, x, a) = nat(c_1) + B(c_1, i', x', a)$	$\{i' = 2*i, x' = x - 1, i \geq 1, x \geq a, i < x\}$

Result: $nat(a\text{-}1)*nat(c_1)$

Listing 8.7. Code of method *logarithmic2*, CRS of the loop at program point 1, and result of the analysis.

As expected, the cost relation system derived from *logarithmic2* looks identical to the one derived from *logarithmic*, up to the fact that the update rule for x (`x' = x – 1`) now reflects the modified behavior of x. However, this time we get $nat(a\text{-}1)*nat(c_1)$ as an answer from the loop cost analysis, which is correct, but much less precise than the bound we got for the loop in method *logarithmic*.

8.7 Nested Loops

We already know method *nested* (listing 8.8) from chapter 6. It contains the typical construct of two nested loops, an inner one iterating from 0 to *b*-1, and an outer one iterating from 0 to *a*-1. In the cost relation system, equation 3) represents the cost of the inner loop. Using equations 3) and 4), a precise upper bound for this cost can be calculated exactly in the same way as in example 8.1. For the outer loop, we also have the usual equations 1) and 2), but in the cost expression of equation 2) we additionally have the term `C(`c_2`, b)`, which reflects one execution of the inner loop in each iteration of the outer loop. For both the inner and the outer loop, we have a simple linear relation as loop condition, as well as linear variable update rules for the involved variables, and PUBS can solve the cost relation system without problems. We get nat(*b*)*nat(c_2) as an upper bound for the cost of the inner loop, and nat(*a*)*(nat(c_1)+nat(*b*)*nat(c_2)) as an upper bound for the cost of the outer loop. Both bounds are precise and tight.

```
action nested(a: Number, b: Number) returns r: Number {
    $r := 0;
    for 0 ≤ i < $a do {              // program point 1
        for 0 ≤ j < $b do {          // program point 2
            $r := $r + 1;
        }
    }
}
```

1) A(c_1, c_2, a, b) = B(c_1, c_2, 0, b, a, a, b)	{true}
2) B(c_1, c_2, j, j_bound, a, i_bound, i, b) = nat(c_1) + B(c_1, c_2, j′, j_bound′, a, i_bound′, i′, b) + C(c_2, b)	{i′ = i + 1, i_bound′ = i_bound, i ≥ 0, i_bound = a, i < i_bound}
3) C(c_2, b) = D(c_5, 0, b, b)	{true}
4) D(c_2, j, j_bound, i, b) = nat(c_2) + M_1(c_2, j′, j_bound′, b)	{j′ = j + 1, j_bound′ = j_bound, j ≥ 0, j_bound = b, j < j_bound}

Result:

nat(*b*)*nat(c_2)	(inner loop)
nat(*a*)*(nat(c_1)+nat(*b*)*nat(c_2))	(outer loop)

Listing 8.8. Code of method *nested*, CRS of the loop at program point 1 (the "outer" loop), and result of the analysis. Note that the CRS of the loop at program point 2 (the "inner" loop) would simply consist of cost relations 3 and 4.

8.8 Conditional inside Loop

Method *if1* (listing 8.9) contains a *for* loop, which in turn contains a conditional. Both the condition of the loop and the condition of the conditional depend on the variable *i*. The loop is iterated as long as *i* (which is initially 0, as we have a *for* loop) is smaller than *a*, which is an argument of the method. The *if*-condition `$i ≥ 2` is false in some loop iterations (namely the first two), such that the *else*-branch of the conditional (which is actually empty) is executed. In the other loop iterations, the *if*-condition is true, such that the *then*-branch of the conditional is executed.

```
action if1(a: Number) returns r: Number {
    for 0 ≤ i < $a do {         // program point 1
        if $i ≥ 2 then {        // program point 2
            $r := $r + 1;
        }
    }
}
```

1) $A(c_1, t_2, f_2, a) = B(c_1, t_2, f_2, 0, a, a)$	$\{true\}$
2) $B(c_1, t_2, f_2, i, i\_bound, a) = nat(c_1) + B(c_1, t_2, f_2, i', i\_bound', a) + C(t_2, f_2, i)$	$\{i' = i + 1, i\_bound' = i\_bound, i \geq 0, i\_bound \geq a, i < i\_bound\}$
3) $C(t_2, f_2, i) = D(t_2, f_2, i) + E(c_{5t}, c_{5f}, i)$	$\{true\}$
4) $D(t_2, f_2, i) = nat(t_2)$	$\{i \geq 2\}$
5) $E(t_2, f_2, i) = nat(f_2)$	$\{i < 2\}$

Result: $nat(a)*(nat(c_1)+nat(t_2)+nat(f_2))$

Listing 8.9. Code of method *if1*, CRS of the loop at program point 1, and result of the analysis.

We again have some kind of nesting here, and in fact this method is handled in the same way as method *nested* in the previous example. In the cost relation system, equation 3) represents the cost of the conditional. For the outer loop, we have the usual equations 1) and 2), where in the cost expression of equation 2) we additionally have the term $C(t_2, f_2, i)$, which represents the execution of the conditional in each iteration of the loop. The loop condition and the *if*-condition both are simple linear relations, and the variable update rules of the involved variables *i* and i*_bound* are linear too, and PUBS is able to solve the cost relation system. We get $nat(a)*(nat(c_1)+nat(t_2)+nat(f_2))$ as an upper bound for the cost of the loop, which is correct, but not very precise, as it takes into account the cost for executing both the *then*-branch (`t₂`) and the *else*-branch (`f₂`) in each loop iteration.

At least, PUBS is able to detect cases where either the *then*-branch or the *else*-branch of the conditional never gets executed. Consider method *if2* (listing 8.10), which is a slight modification of method *if1*, where we changed the *if*-condition to `$i ≥ 0`. Obviously, the *else*-branch of the conditional is never executed, as `$i ≥ 0` is an invariant of the loop. In this case, we get nat(a)*(nat(c_1)+nat(t_2)) as an upper bound for the cost of the loop, which is a correct and precise bound.

```
action if2(a: Number) returns r: Number {
    for 0 ≤ i < $a do {         // program point 1
        if $i ≥ 0 then {        // program point 2
            $r := $r + 1;
        }
    }
}
```

1) $A(c_1, t_2, f_2, a) = B(c_1, t_2, f_2, 0, a, a)$	{true}
2) $B(c_1, t_2, f_2, i, i\_bound, a) = nat(c_2) + B(c_1, t_2, f_2, i', i\_bound', a) + C(t_2, f_2, i)$	$\{i' = i + 1, i\_bound' = i\_bound, i \geq 0, i\_bound \geq a, i < i\_bound\}$
3) $C(t_2, f_2, i) = D(t_2, f_2, i) + E(c_{5t}, c_{5f}, i)$	{true}
4) $D(t_2, f_2, i) = nat(t_2)$	$\{i \geq 0\}$
5) $E(t_2, f_2, i) = nat(f_2)$	$\{i < 0\}$

Result: nat(a)*(nat(c_1)+nat(t_2))

Listing 8.10. Code of method *if2*, CRS of the loop at program point 1, and result of the analysis.

8.9 Composite Loop Condition

So far we only looked at loops which have a basic linear relation as loop condition. Our loop analysis can additionally handle loops with a loop condition which is any nested structure of conjunctions, disjunction and negations of such basic relations.

```
action and1() {
    $i := 0;
    $j := 0;
    while $i < 10 and $j < 20 do {        // program point 1
        $i := $i + 1;
        $j := $j + 3;
    }
}
```

1) $A(c_1) = B(c_1, 0, 0)$	{true}
2) $B(c_1, i, j) = C(c_1, i, j)$	{true}
3) $C(c_1, i, j) = D(c_1, i, j)$	$\{i < 10\}$
4) $D(c_1, i, j) = nat(c_1) + B(c_1, i', j')$	$\{i' = i + 1, j' = j + 3, i \geq 0, j \geq 0, j < 20\}$

Result: 22/3*nat(c_1)

Listing 8.11. Code of method *and1*, CRS of the loop at program point 1, and result of the analysis.

Method *and1* (listing 8.11) executes a loop which has a conjunction of two linear relations as loop condition. The loop is iterated as long as both `$i < 10` and `$j < 20` hold. Both *i* and *j* are increased linearly in each iteration of the loop. Method *and2* (listing 8.12) is semantically equivalent to *and1*, but when we look at its source code we see that the order of the two basic linear relations is the other way round than in *and1*, which also leads to a slightly different cost relation system (see listing 8.12), in which the order of variables *i* and *j* is reversed compared to the cost relation system for the loop in method *and1* (see listing 8.11). When computing an upper bound for the cost of the loops in *and1* and *and2*, respectively, we can simply ignore one of the two operands of the conjunction. As we are looking for an overapproximation of the loop cost, we still get a correct result with this approach, at the cost that the bound we get might be not precise. It seems that this is exactly what PUBS does in order to solve the cost relation systems of this example. We see this by looking at the results we get from the loop cost analysis: For the cost of the loop in *and1*, we get the correct and tight bound 22/3*nat(c_1) as result, while for the cost of the loop in *and2*, we get 10*nat(c_1), which is correct but not precise.

```
action and2() {
    $i := 0;
    $j := 0;
    while $j < 20 and $i < 10 do {       // program point 1
        $i := $i + 1;
        $j := $j + 3;
    }
}
```

1) $A(c_1) = B(c_1, 0, 0)$	{true}
2) $B(c_1, j, i) = C(c_1, j, i)$	{true}
3) $C(c_1, j, i) = D(c_1, j, i)$	$\{j < 20\}$
4) $D(c_1, j, i) = nat(c_1) + B(c_2, j', i')$	$\{j' = j + 3, i' = i + 1, j \geq 0, i \geq 0, i < 10\}$

Result: 10*nat(c2)

Listing 8.12. Code of method *and2*, CRS of the loop at program point 1, and result of the analysis.

In method *or* (listing 8.13) we have a loop with a disjunction of two linear relations as loop condition. As soon as a disjunction is involved somewhere in the loop condition, the loop cost analysis is almost never able to find an upper bound for the cost of the loop, due to PUBS which has problems handling such cases (it can not find a ranking function). *or* is one such example. Even though we have simple linear relations on both sides of the *or* in the loop condition, and the variable update rules for *i* and *j* are available and linear, we do not get an upper bound as an answer from the analysis.

```
action or() {
    $i := 1;
    $j := 0;
    while $i < 10 or $j < 50 do {       // program point 1
        $i := $i + 1;
        $j := $j + 2;
    }
}
```

1) $A(c_1) = B(c_1, 1, 0)$	$\{true\}$
2) $B(c_1, i, j) = C(c_1, i, j)$	$\{true\}$
3) $B(c_1, j, i) = D(c_1, i, j)$	$\{true\}$
4) $C(c_1, i, j) = nat(c_1) + B(c_1, i', j')$	$\{i' = i + 1, j' = j + 2, i \geq 1, j \geq 0, i < 10\}$
5) $D(c_1, i, j) = nat(c_1) + B(c_1, i', j')$	$\{i' = i + 1, j' = j + 2, i \geq 1, j \geq 0, j < 50\}$

Result: no upper bound found

Listing 8.13. Code of method *or*, CRS of the loop at program point 1, and result of the analysis.

8.10 Variable Update Depending on Other Variable

So far, we only looked at examples where the update rule for some variable (e.g. *i*) had the form `i' = a*i + b`, where *a* and *b* are rational constants. But we are not restricted to this, as long as the variable update rules are linear.

We already know method *sumOfSums* (listing 8.14) from before. It is an example where the current value of some variable determines how some other variable is updated in each iteration of the loop. *sumOfSums* contains a loop which is iterated as long as *x* (which is initially 1), is smaller than some constant *a*, which is given as an argument of the method. In the loop we have an auxiliary variable *i*, whose value is initially 1 and is increased by 1 in every loop iteration. In each iteration of the loop, the value of *x* is increased by the current value of *i*. This gives us the variable update rule `x' = x + i` for *x* in the cost relation system created by the loop cost analysis. The condition of the loop is a simple linear relation, and the update rules for *i* and *x* are linear as well, and PUBS is able to solve the cost relation system. We get nat(a-1)*nat(c_1) as an upper bound for the cost of the loop, which is correct but not precise.

```
action sumOfSums(a: Number) returns r: Number {
    $x := 1;
    $i := 1;
    while ($x < $a) do {        // program point 1
        $i := $i + 1;
        $x := $x + $i;
    }
    $r := $i;
```

1) $A(c_1, a) = B(c_1, 1, a, 1)$	$\{true\}$
2) $B(c_1, x, a, i)$ $= nat(c_1) + B(c_1, x', a, i')$	$\{x' = x + i, i' = i + 1, x \ge 1, i \ge 1, x < a\}$

Result: nat(a-1)*nat(c_1)

Listing 8.14. Code of method *sumOfSums*, CRS of the loop at program point 1, and result of the analysis.

9 Case Studies from the TouchDevelop Cloud

In this chapter, we present the result of the loop cost analysis for some selected scripts from the TouchDevelop cloud. In the code listings in this chapter, we replaced parts of the code which are not relevant for the loop cost analysis (e.g. parts of long string literals, implementations of called methods not containing loops themselves, etc.) by the notation *[...]*. The original code of a script can be found at

```
http://www.touchdevelop.com/api/<script_id>/text
```

where <*script_id*> must be replaced by the four-letter identifier of the corresponding script (given in the titles of the following sections).

9.1 Coding Duel (csbl)

Listing 9.1 shows the code of a simple number guessing game. It is a typical example for a script from the TouchDevelop cloud, as it is rather small (72 lines of code) and uses only few standard methods. The script contains two loops. Loop 5 is a typical *for* loop, simply having a constant as the upper bound for the iteration variable. This type of loop represents the kind of loop that is most commonly used in scripts published in the cloud. The loop cost analysis finds 10*nat(c_5) as an upper bound for the cost of loop 5, which is both correct and precise.

The other loop in this script, which we denote by loop 1, is somewhat more complicated. It contains a construct of three nested conditionals. Still, the loop is easy to handle, and the loop cost analysis gives us 1000*(c_1 + t_2 + f_2 + t_3 + f_3 + t_4 + t_4) as an upper bound for the cost of this loop. This example illustrates nicely the purpose of the loop cost analysis as it is presented in the introduction of this thesis: The factor of 1000 in the resulting cost expression indicates that the cost of executing loop 5 might actually be rather high. If this is the case, it might be a good idea to execute the script in the TouchDevelop cloud instead of running it locally on the smartphone.

```
meta name "coding duel";
// Can you discover the secret program? [...]

action main() {
  $success_count := 0;
  $failure_count := 0;
  $sample_count := 5;
  for 0 ≤ x < 1000 do {                                // program point 1
    $secretx := code→secret($x);
    $px := code→puzzle($x);
    $msg := "x := " ∥ $x ∥ " ==> secret :=
      " ∥ $secretx ∥ ", puzzle := " ∥ $px;
    if $px ≠ $secretx then {                           // program point 2
      $failure_count := $failure_count + 1;
      if $failure_count < $sample_count then {         // program point 3
        code→create_error($msg);
      }
    }
    else {
      $success_count := $success_count + 1;
      if $success_count < $sample_count then {         // program point 4
        code→create_success($msg);
      }
    }
  }
  if $failure_count > 0 then {
    phone→vibrate(0.1);
    "Try again...\n [...] " →post_to_wall;
  }
  else {
    for 0 ≤ i < 10 do {                                // program point 5
      "You won!" →post_to_wall;
    }
  }
}

action secret(x: Number) returns r: Number { [...] }

action puzzle(x: Number) returns r: Number { [...] }

var error : Color { [...] }

action create_error(msg: String) { [...] }

var success : Color { [...] }

action create_success(msg: String) { [...] }
```

Listing 9.1. Code of script **csbl** (“coding duel”).

9.2 Password Generator (hbtn)

In listing 9.2 we see the code of a password generation application. When running the script, the user can first select one of six predefined password patterns from a list of strings. Then, the script generates a random password according to the user's choice and stores it in a table on the phone. There is a single loop in this script (loop 1), in which the password gets assembled according to the chosen pattern. The loop directly contains six conditionals. In each of them, one class of characters that may be used in a password (e.g. uppercase consonants or lowercase vowels) is handled. It is easy for the loop cost analysis to compute an upper bound for the cost of the loop: We get $i\_bound*(c_1 + t_2 + f_2 + t_3 + f_3 + t_4 + f_4 + t_5 + f_5 + t_6 + f_6 + t_7 + f_7)$ as an answer. Here, we can see a weakness of the loop cost analysis. The cost expression we get contains i_bound, which does not appear in the original TouchDevelop program. Instead, we would prefer to have something like *Length(patterns→at(x))* here. This happens because for the program point just before the loop, the loop cost analysis does not get a linear relation of the form *i_bound* = *expr* (for some linear expression *expr*) from the result of the numerical analysis. The absence of such a constraint is not surprising, as Sample tries to evaluate method calls such as *patterns→at(x)*, which is not possible statically, as *x* (the index of the pattern chosen by the user) is only known at runtime. So the analysis cannot know to which of the six possible strings *patterns→at(x)* actually refers, and consequently it can not know to the length of which string *i_bound* is equal. Also, the loop cost analysis cannot know that the variable *i_bound* does not appear in the original TouchDevelop code, as it operates on the CFG of the *main()* method (where *i_bound* appears, as it was introduced when the for loop was converted into a control flow graph loop). But even if it would know it, it would not be possible to replace it by some other expression, because we do not know the initial value of *i_bound*, as explained before. So – at least for the moment – we have to live with the fact that a cost expression returned by the loop cost analysis may contain a variable name that does not appear in the input script. However, with a better abstraction we could solve this problem.

```
meta name "Password Gen Pro";
// Generate unique passwords with the ability to change the pattern. [...]

action main() {
  wall→set_subtitle("Generate a unique password. [...] ");
  data→c := "aeiou";
  data→v := "bcdfghjklmnpqrstvwxyz";
  data→n := "1234567890";
  data→s := "$%&";
  $patterns := collections→create_string_collection;
  $patterns→add("vCnscV");
  $patterns→add("cVsnCv");
  $patterns→add("nVcsvC");
  $patterns→add("VCsnVnvC");
  $patterns→add("cVnsvnCv");
  $patterns→add("nCvsVnvv");
  $x := wall→pick_string("Pick a password pattern", "c = consonants, v =
    vowels, n = numbers and s = special characters. \n Uppercase version of
    above means just that.", $patterns);
  $password := collections→create_string_collection;
  // Loop password characters
  for 0 ≤ i < $patterns→at($x)→count do {              // program point 1
    $char := $patterns→at($x)→substring($i, 1);
    if $char→equals("c") then {                        // program point 2
      $password→add(data→c→substring(math→random(data→c→count), 1));
    }
    if $char→equals("C") then {                        // program point 3
      $password→add(data→c→substring(math→random(data→c→count),  1)
      →to_upper_case);
    }
    if $char→equals("v") then {                        // program point 4
      $password→add(data→v→substring(math→random(data→v→count), 1));
    }
    if $char→equals("V") then {                        // program point 5
      $password→add(data→v→substring(math→random(data→v→count), 1)
      →to_upper_case);
    }
    if $char→equals("n") then {                        // program point 6
      $password→add(data→n→substring(math→random(data→n→count), 1));
    }
    if $char→equals("s") then {                        // program point 7
      $password→add(data→s→substring(math→random(data→s→count), 1));
    }
  }
  $h := records→history_table→add_row;
  $h→password→set($password→join(""));
  $h→generated→set(time→now);
  $password→join("")→post_to_wall;
  $password→join("")→copy_to_clipboard;
  wall→prompt("Password copied to clipboard.");
}

var c : String { } var v : String { }
var n : String { }
var s : String { }
table history { [...] }
```

Listing 9.2. Code of script **hbtn** (“Password Gen Pro”).

9.3 Shakespearian Insults Generator (nlqo)

Published TouchDevelop scripts are often not written in the most optimal way, which is due to the fact that many TouchDevelop users are novice programmers or hobbyists. See for example the script in listing 9.3. It implements some kind of phrase generator, which randomly combines adjectives out of three lists (of which two are identical, probably by mistake) to create a "Shakespearian Insult". The script contains three almost identical *while* loops. In each loop, there is a variable *len*, and the loop is iterated as long as *len* is non-negative. *len* is initially 0, so the loop body is executed a first time. In this first loop iteration, *len* is decreased by one, so it becomes -1, and the loop terminates. So we can see that these loops are actually superfluous (in the sense that the statements of the loop body could directly be written in the code, without being inside a loop), as their bodies are executed exactly once. Although this script is rather simple, it provides an interesting test case for the loop cost analysis. In fact, we correctly get nat(c_1) as an upper bound for the cost of loop 1 (and, analogously, nat(c_2) and nat(c_3) for the other two loops), so we see that the loop cost analysis is able to correctly handle this border case.

```
meta name "Shakespearian Insults Generator";

action Generate() {
  wall→set_background(colors→chrome);
  wall→prompt("To generate a new insult, refresh the page. [...] ");
  $s := "artless, bawdy, beslubbering, bootless, [...] , yeasty\n";
  $pass := "";
  $len := 0;
  while $len ≥ 0 do {                                 // program point 1
    $charIndex := math→random(50);
    $pass := $pass ∥ $s→split(",")→at($charIndex);
    $len := $len - 1;
  }
  code→Generate2;
  $pass→post_to_wall;
  "Precede each insult with thou." →post_to_wall;
}

action Generate2() {
  $CharMap := "base-court, bat-fowling, [...] , weather-bitten,";
  $pass := "";
  $len := 0;
  while $len ≥ 0 do {                                 // program point 2
    $charIndex := math→random(49);
    $pass := $pass ∥ $CharMap→split(",")→at($charIndex);
    $len := $len - 1;
  }
  code→Generate3;
  $pass→post_to_wall;
}

action Generate3() {
  $charMap := "base-court, bat-fowling, [...] , weather-bitten,";
  $pass := "";
  $len := 0;
  while $len ≥ 0 do {                                 // program point 3
    $charIndex := math→random(49);
    $pass := $pass ∥ $charMap→split(",")→at($charIndex);
    $len := $len - 1;
  }
  $pass→post_to_wall;
}
```

Listing 9.3. Code of script **nlqo** (“Shakespearian Insults Generator”).

9.4 Text my Location (moji)

In listing 9.4 we see a simple script which first asks the user to choose a phone number on his smartphone. Then, it sends a short message – containing information about the current location of the phone – to the selected phone number. The *main()* method of the script contains a single *while* loop (loop 1). When we use the script as input for the loop cost analysis, it can not find an upper bound for the cost of loop 1. Since the loop condition `data→link→is_invalid` may always be false (if the user never enters a valid phone number), it is possible that the loop does not terminate.

```
meta name "text my location";
// Send your address and location through messaging

action main() {
  // Grabs location and sends it by sms
  // get a phone number and cache it
  while data→link→is_invalid do {                    // program point 1
    data→link := phone→choose_phone_number;
  }
  // get the current location
  $loc := senses→current_location_accurate;
  $address := locations→describe_location($loc);
  data→main_tile→set_back_title($address);
  social→send_sms(data→link→address, "we are at " ‖ $address ‖ ", " ‖
$loc→to_string);
}

var main_tile : Tile {
  readonly = true;
}

var link : Link {
}
```

Listing 9.4. Code of script **moji** (“text my location”).

We observed that several *while* loops found in scripts from the TouchDevelop cloud have the form while(*true*) or while*(“user input is not valid”)*. For the cost of all these loops, our analysis can not find an upper bound, as obviously non-termination is possible for such loops.

10 Experimental Results

To assess to quality of our implementation, we ran the loop cost analysis on a large number of real scripts from the TouchDevelop cloud. We only considered scripts which actually contain loops, and we only considered root scripts. Remember that a root script is a script that is not a modified version of some other script. We decided to use only root scripts because usually these scripts already contain all the loops which appear in the scripts derived from them. So analyzing all scripts (including derived scripts) would not give us a representative result since we would re-analyze the same loops (or even the same, identical scripts) many times. We used a timeout of 5 minutes when analyzing the scripts.

We ran the analysis on a machine with the following specification: Intel Core 2 Quad CPU Q9550 2.83GHz, 4 GB RAM, Ubuntu 12.04 LTS 32 bit, Java SE Runtime Environment 1.7.0_17-b02.

Overall, we analyzed 1737 scripts. The analysis could compute a non-zero upper bound of the cost of 2144 loops. For 979 loops, the analysis returned 0 as an upper bound. For 1241 scripts it could not find an upper bound of the cost. Note that the analysis did actually not fail in many of these cases because of the following points:

In the cases where the analysis returned 0, the Sample analysis (whose result is used by the loop cost analysis) inferred a bottom state for the program points inside the loop, which means that they can never be reached. Using this fact, the loop cost analysis directly returns 0 in such cases, without needing to create a cost relation system and calling PUBS. We can distinguish two cases:

- It might be that the corresponding loop is never executed, either due to a faulty implementation of the input program (which actually occurs quite often, as many TouchDevelop programmers are beginners or hobbyists), or due to full intention of the programmer. For example, a programmer might use a `while(false)` loop instead of commenting out pieces of code, because (as we observed) there seems to be no easy way to comment out code when using the TouchDevelop programming environment.

- In all other cases where the analysis returned 0, the Sample analysis wrongly inferred bottom for the program points inside the loop. This is due to parts of the semantics of TouchDevelop which are currently under-defined in Sample. So, in these cases, it is not the part of the loop cost analysis implemented in this thesis that fails, but a part of Sample on which our implementation relies.

Currently, the TouchDevelop semantics in Sample is being extended and tested, and we expect to fix this issue in the next few weeks.

In the cases where the analysis could not find an upper bound of the cost, we can analogously distinguish two cases:

- In some (probably most) cases where the analysis could not find an upper bound of the cost of the loop, there might indeed exist no upper bound. We saw in section 9.4 that this is the case for a lot of while loops in real TouchDevelop, such as loops of the form `while(true) { … }`.

- In other cases where the analysis could not find an upper bound, it might in fact be the case that the cost of the loop is bounded, but our analysis failed in finding an upper bound. As most loops in TouchDevelop tend to have a rather simple structure, we assume that such cases were rare.

When developing the implementation of the loop cost analysis, we did not pay attention to optimizing its performance with respect to runtime, memory consumption, and so on. Still, we would like to present some numbers here, which we measured when running the analyses on the set of 1737 scripts. Overall, all these analyses together took:

- 834 seconds for compiling (including the time required to download the scripts and to augment the control flow graphs)
- 5616 seconds for the numerical analysis
- 1063 seconds for calculating the loop costs (including the generation of the cost relation systems, and solving the cost relation systems using PUBS)

This means that in average for one script the analysis took:

- 0.48 seconds for compiling
- 3.23 seconds for the numerical analysis
- 0.61 seconds for calculating the loop costs

These results show that the analysis is precise and scales up in practice.

11 Related Work

In this chapter, we present some work that is related to this thesis. First, we have a look at an approach to the cost analysis of object-oriented bytecode programs. Then, we discuss an approach for solving cost relation systems. PUBS is an implementation of this approach.

11.1 Cost Analysis of Object-Oriented Bytecode Programs

Developing a precise overapproximation of the cost of a program is a complex problem that has been (partially) explored in the last few years. Albert et al. [2] present the first approach to the automatic cost analysis of object-oriented bytecode programs. Their method takes a bytecode program and a cost model specifying the resource of interest as input, and returns a set of recursive equations, which capture the execution cost of the program. In a first step, their approach generates an intermediate *rule-based representation (RBR)* from the original bytecode. Then, they use static analysis to infer linear *size relations* among program variables at different program points. As a size abstraction for integer variables they use the value of the variable, whereas the size abstraction of a data structure $x \in dom(lv)$ (where *lv* denotes some variable mapping) with respect to some heap is defined as the length of the maximal path reachable from the reference *lv(x)* by dereferencing, i.e., following other references as fields. The path-length of `null` is defined to be 0, that of a cyclic data structure is defined to be ∞. The next step consists of finding an appropriate cost model, which defines how cost is assigned to each execution step (for example, a cost model counts the number of instructions or the amount of memory consumption). Then, their method generates a cost relation system from the RBR, the size relations and the cost model.

The approach described here often relies on unsound assumptions. For instance, Albert et al. ignore cyclic data structures. When approximating the cost of a program, it is crucial to find an appropriate abstraction of the heap. A *maximal path-length* abstraction as used in their approach might yield good results in some practical cases, but in other cases it might be unsound or imprecise. For example, consider a cyclic data structure, which per definition has a size (i.e., maximal path-length) of ∞. However, it might be the case that in every execution of a given program, this data structure is traversed in a controlled way such that the actual maximal length of any path ever taken is some finite number *n*. In such a case, using the *maximal path-length* abstraction would yield a rather imprecise overapproximation of the cost of the program.

11.2 Closed-Form Upper Bounds in Static Cost Analysis

Albert et al. [1] present an approach for obtaining closed form upper bounds for cost relation systems. PUBS (section 3.2) is an implementation of this approach. Here, we present a summary of the main ideas of the approach.

Given a cost relation system *S*, the approach first infers a set of so-called *evaluation trees*, where each such tree represents a possible strategy to solve *S*. Given an evaluation tree *T*, the sum of all nodes in *T* corresponds to the result of solving *S* using the evaluation strategy represented by *T*. PUBS then tries to find the largest cost one may get from evaluating *S* using any evaluation tree, which is exactly the upper bound it is looking for. Note that this cost is not always computable, as there might be infinite evaluation trees. Next, Albert et al. present an approximation scheme to actually infer the closed-form upper bounds. It is based on the idea of bounding the cost of the corresponding evaluation trees. To do this for some evaluation tree *T*, their approach computes upper bounds for both the number of nodes of *T*, as well as for the cost of the nodes of *T*.

Given an evaluation tree *T* for a cost relation *C*, the number of nodes in *T* can be derived from the *depth* of *T* and the *branching factor* of *T*. At this point, Albert et al. introduce the notion of a *loop* in a cost relation *C*, which is used to model consecutive calls of *C*. Intuitively, a loop $C(\bar{v}_1) \rightarrow C(\bar{v}_2)$ means that evaluating $C(\bar{v}_1)$ may eventually be followed by an evaluation of $C(\bar{v}_2)$. In an evaluation tree this means that the node corresponding to $C(\bar{v}_1)$ has a child corresponding to $C(\bar{v}_2)$. Next, the paper introduces a specific form of *ranking functions*: A function is a ranking function for a cost relation *C* if it is a ranking function for all loops in *C*. Their approach uses such ranking functions as an upper bound of consecutive calls (and therefore on the height of the corresponding evaluation trees). This is justified by the facts that the ranking function decreases by at least 1 in each iteration and that it is always non-negative. Intuitively, if their method can find a ranking function for a cost relation *C*, then it is able to compute an upper bound for the cost represented by *C*. The current implementation of the approach, PUBS, is restricted to linear ranking functions. To bound the cost of the nodes in an evaluation tree *T*, their approach relies on loop invariants: Given a cost relation $\langle C(\bar{v}) = exp + \sum_{i=1}^{k} D_i\left(\bar{w}_i\right),\ P\rangle$, the approach tries to find an invariant, in terms of linear constraints, that holds between the arguments at the initial call of *C*, and the arguments at each consecutive call of *C* during the evaluation of the initial call. Given *C* and a safe approximation of its loop invariant, their approach can now compute an upper bound for *exp* by maximizing its `nat` components. Finally, Albert et. al present an extension of the basic approach which may help to obtaining more precise upper bounds for divide and conquer programs. It is based on counting *levels* in an evaluation tree rather than counting nodes.

12 Conclusion

We successfully developed and implemented an analysis which overapproximates the cost of TouchDevelop loops. We implemented the loop cost analysis as an extension of the static analyzer Sample. This gives us the great advantage that we can use existing heap and value analyses of Sample – in particular a numerical analysis that uses Apron – to support our loop cost analysis with linear constraints that hold at specific program points in the program under examination. We found a way to use this information to create a system of cost relations which represents an overapproximation of the cost of the loop. To solve such cost relation systems, we included a call to the solver PUBS in our implementation. This finally gives us a closed form upper bound of the cost of the loop under examination.

By running the analysis on a series of test input scripts constructed by ourselves, as well as hundreds of real scripts on the TouchDevelop cloud, we observed that our analysis produces satisfying results in most cases, and it scales up. The loop cost analysis uses several existing tools and libraries (Sample, Apron, and the PUBS solver), so the result of running the analysis strongly depends on the possibilities and limits of these:

- The numerical analysis we run in Sample, which is used by the loop cost analysis to infer the cost relation system, plays a central role. First of all, the loop cost analysis relies on its soundness and precision. Secondly, the loop cost analysis depends on getting the "right" constraints from the numerical analysis, i.e., the ones that it can use to generate parts of the cost relation system. For instance, an equality of the form $i = old\_i + 1$ (for some loop variable *i*) may be very helpful for the loop cost analysis (as we saw in chapter 7), while some other constraint $exp > 0$ might not be needed at all by the loop cost analysis because no variable occurring in the expression *exp* has an influence on the number of times that the loop under examination is iterated.

- Having a cost relation system which captures the cost of a loop does not guarantee that we will get a precise bound of the cost of this loop. The quality of our analysis results depends on the performance of PUBS when it tries to solve this cost relation system. For instance, we saw in chapter 8 that PUBS yields precise results for cost relation systems derived only from 'additive' variable update rules of the form $i = old\_i + expr$. However, as soon as a cost relation system additionally contains linear relations derived from variable update rules of a 'multiplicative' kind, such as $i = n*old\_i + expr$, we might get a less precise answer from PUBS, as we saw for instance in section 8.6.

Finally, we would like to think about possible future extensions of our work. A major point is the cost model. Currently, the loop cost analysis does not use a concrete cost model, but instead uses symbolic cost parameters. So the result of the analysis is an abstract cost expression in terms of these cost parameters. The main disadvantage of such an abstract result is the fact that it can not (or – at least – not directly) be used in practice. A possible extension of this master thesis would be to develop a more specific cost model which involves actual values rather than symbolic parameters. For instance, such cost values might be the number of statements in a control structure, or the memory or energy consumption needed to execute a TouchDevelop instruction (or a group of them). Such a cost model would have the following advantages:

- The analysis would return a concrete value (e.g. an amount of energy). This statically determined value could then be compared with a real, measured value in order to check the correctness of the analysis. With our parameterized cost model, we did not have this possibility.

- The loop cost analysis could be used in practice, e.g. to estimate the worst-case energy consumption when running a script. As stated in the introduction of this master thesis, we could then attach this cost information to the script and use it at runtime to decide whether the script should be executed locally on the smartphone, or in the TouchDevelop cloud.

Another possible extension of our work would be to develop a custom-made numerical analysis. Currently, the loop cost analysis uses a generic numerical analysis (Apron linear equalities) which is available in Sample, and which was not adapted in any way to the needs of the loop cost analysis. As discussed above, the loop cost analysis depends on getting the “right” linear constraints from the numerical analysis. With a non-adapted numerical analysis, the loop cost analysis sometimes receives (possibly non-linear) constraints which are not helpful at all, while other, helpful constraints might be missing, as they could not be inferred by this particular numerical analysis. So developing and implementing a new numerical analysis (most probably based on existing work) which is custom-made for the needs of the loop cost analysis might help to improve the performances and the quality of the result of the loop cost analysis.

References

[1] E. Albert, P. Arenas, S. Genaim, G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. In *Journal of Automated Reasoning, 46 (2), pp. 161–203*, 2011.

[2] E. Albert, P. Arenas, S. Genaim, G. Puebla and D. Zanardini. Cost Analysis of Object-Oriented Bytecode Programs. In *Journal of Theoretical Computer Science, 413 (1), pp.142-159*, 2012.

[3] G. Costantini, P. Ferrara and A. Cortesi. Static Analysis of String Values. In *Proceedings of ICFEM '11, pp. 505-521*, 2011.

[4] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of POPL '77*. ACM Press, 1977.

[5] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proceedings of POPL '79*. ACM Press, 1979.

[6] P. Ferrara. Static Type Analysis of Pattern Matching by Abstract Interpretation. In *Proceedings of FMOODS '10, pp. 186–200*, 2010.

[7] P. Ferrara, R. Fuchs and U. Juhasz. TVAL+: TVLA and Value Analyses Together. In *Proceedings of SEFM '12, pp. 63-77*, 2012.

[8] P. Ferrara and P. Müller. Automatic Inference of Access Permissions. In *Proceedings of VMCAI '12, pp. 202-218*, 2012.

[9] N. Horspool, J. Bishop, A. Samuel, N. Tillmann, M. Moskal, J. de Halleux and M. Fähndrich. TouchDevelop – Programming on a Phone. Version 1.1 for TouchDevelop 2.8, May 2012.

[10] B. Jeannet and A. Miné. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *Proceedings of CAV '09, pp. 661-667*, 2009.

[11] M. Karr: Affine Relationships among Variables of a Program. In *Acta Informatica, pp. 133-151*, 1976.

[12] M. Zanioli, P. Ferrara and A. Cortesi. SAILS: Static Analysis of Information Leakage with Sample. In *Proceedings of SAC '12, pp. 1308-1313*, 2012.

[13] Breadth-first search. http://en.wikipedia.org/wiki/Breadth-first_search, accessed on April 6, 2013.

[14] Introduction to Records in TouchDevelop. http://az31353.vo.msecnd.net/cpd/ xeuo-records.pdf, accessed on April 6, 2013.

[15] TouchDevelop. http://research.microsoft.com/en-us/projects/touchdevelop/, accessed on April 6, 2013.

[16] Windows Phone. URL: http://www.microsoft.com/windowsphone/, accessed on April 6, 2013.