Inferring counterexamples from abstract error states

Raphael Fuchs
fuchsra@student.ethz.ch

Master's thesis report

Chair of Programming Methodology
Department of Computer Science
ETH Zurich

April 7, 2014

Supervised by:
Lucas Brutschy
Prof. Dr. Peter Müller

Chair of Programming Methodology

inf | Informatik
Computer Science

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

TouchDevelop is a beginner-friendly programming language and platform enabling smartphone users to write scripts in a mobile environment. It was developed by Microsoft Research and integrates closely with Microsoft cloud services. As scripts often contain errors that may cause the program to crash or misbehave during execution, the TouchBoost project applies static analysis techniques to detect such possible errors ahead of execution time. However, since the behavior of the programs has to be over-approximated, there can be false alarms and it may be unclear how the program reaches a potential error.

In this Master's thesis we enhance the TouchBoost static analyzer with backward analysis functionality [Riv05]. The approach based on abstract interpretation allows us to narrow down entry states that lead to potential errors. We then use these refined entry states to infer concrete counterexamples. When executed, such a counterexample leads to the abstract error reported by the forward abstract interpreter and helps the user better understand the problem at hand. In some situations, the technique can also prove alarms to be false. We demonstrate that the approach is effective for a large set of published TouchDevelop scripts.

Contents

1 Introduction

1.1 Motivation and Goals

When a static analyzer like TouchBoost finds a potential error, the analysis usually reports the problem in a form similar to

“`Assertion` *`expr`* `may not hold at program point` *`pp`*`.`” (1)

A typical error is to call a method on an invalid reference: E.g., in `obj->method(param)`, the analyzer raises an error if it cannot prove that the implicitly generated assertion `obj != invalid` holds in every execution of the program. Other common errors include passing of invalid arguments to methods that do not expect them, violating numerical bounds, and accessing resources without checking whether they are available.

However, a problem is often only triggered by a particular subset of all possible inputs. In many cases it is not obvious how the reported errors relate to interactive inputs and program state such as the values of method arguments and global variables at the beginning of the method containing the error. There is also the possibility of false (spurious) alarms because the analysis is conservative and never misses any potential errors, at the cost of considering program behavior that may never actually happen.

The goal of this thesis is to improve the error reporting in TouchBoost. We narrow down the conditions that must be fulfilled at the program entry to reach the point where a potential error is reported. If we detect that no actual inputs can satisfy the constraints, we classify the alarm as false and drop it. Otherwise, we try to synthesize concrete program inputs for the user that cause the error to occur.

The action in Listing 1 serves as our motivating example. It is a modified version of an action found in the TouchDevelop script “TextMaster” (id `hwyo`). The action creates an artistic image by drawing text multiple times with random position, orientation and color. Two problems for which alarms are produced exist in its code, both related to violated numerical bounds:

- On line 5, a picture of some requested size is created. However, the dimensions are directly taken from user input, without any validation. The user may enter a negative number, or the number string may fail to parse and turn into NaN. Such numbers certainly do not satisfy any preconditions the method `create picture` may reasonably have.
- A more subtle bug is present on line 11 when calling the method “`draw text`”: Both the x and y-coordinates of the drawn text are chosen at random from $[0, w]$. The author meant to write `rand(h)` to generate a pseudo-random y-coordinate, as the text may sometimes be drawn vertically outside the picture bounds in the erroneous version.

To narrow down the conditions leading to a given error, we start at its program location, assuming the assertion is indeed violated. A *backward analysis* then reasons backwards toward the program entry and keeps track of constraints. For example, the second bug above is only reached when the *radial* boolean variable is $false$. Its value in turn originates from interactive user input in the program GUI. Furthermore, the random y-coordinate must be smaller than the entered width of the picture but at the same time greater than the entered height. When we are not sure whether a constraint always holds, we do not assume it. Hence, the analysis over-approximates and returns *necessary*, but not *sufficient* conditions for the error.

At the program entry, the backward analysis results in the abstract constraints we have just described. We then try to produce a concrete counterexample satisfying the constraints, e.g. for the second bug we may suppose that the user inputs a width of 400, height 200, selects the option flag to be $false$ and the random coordinates happen to be $x = 300$, $y = 600$. Since our conditions are not sufficient, we perform testing on that input instance to ensure that it actually triggers the error. The testing is done by running the method in a TouchDevelop interpreter.

```
action draw(text: String, font: String) {
  var w := wall->ask number("width?")
  var h := wall->ask number("height?")
  var radial := wall->ask boolean ("radial shape?")
  var pic := media->create picture(w, h)
  for 0 <= i1 < 50 do {
    if radial then {
      pic->draw text(w/2, h/2,
            text, font, math->rand(360), colors->rand)
    } else {
      pic->draw text(math->rand(w), math->rand(w),
            text, font, 0, colors->rand)
    }
  }
}
```

Listing 1: Motivating example

1.2 Outline

We first give some background on both TouchDevelop and TouchBoost in Section 2. The approach we take is explained in Section 3, together with some foundations of abstract interpretation and their formal description. Section 4 describes how the backward analysis was implemented in TouchBoost, detailing changes to abstract domains and implementation obstacles we faced. In Section 5 we explain how the abstract states obtained from the backward analysis are concretized and then used as inputs for concrete testing. Section 6 demonstrates a few results of the analysis. Finally, in Section 7, we summarize our findings, discuss related work on the topic and point out possible future improvements. Appendix B briefly lists some other work done during the thesis.

2 Background

2.1 TouchDevelop

The TouchDevelop programming environment [Mic, TMdHF11], developed by Microsoft Research, enables programmers to create scripts for their smartphones. Applications are written directly on the devices, and they have access to all functionality common in a mobile environment: Music and images on the device, sensors, cloud services and social networks.

The language itself statically typed and has no advanced subtyping mechanisms or parametric polymorphism. It tries to hide technicalities as much as possible. For example, there is a single number type instead of separate integer and floating point types. Additionally, type inference for local variables enables easier script input with less overhead.

TouchDevelop's execution model is traditionally imperative with global mutable state. The execution is not multi-threaded; no concurrent interleavings of instructions may occur. Code is composed into *actions* and *events*. Events are run after actions in response to user inputs, on completion of long-running operations etc. Scripts can be executed on multiple execution platforms: A variety of web browsers and smartphone OS like Windows Phone 8 are supported.

2.2 Sample and TouchBoost

A lot of published TouchDevelop scripts contain errors which may cause the program to crash or misbehave during execution. One reason might be that TouchDevelop mainly targets hobbyists, many of whom write program code for the first time. Furthermore, the one-off nature of many of these scripts contributes to omitted error checking.

The TouchBoost project attempts to increase the reliability of scripts and detect possible errors by employing static analysis techniques. It utilizes the Sample framework (Static Analysis of Multiple Languages), which is a generic static analyzer based on abstract interpretation [CC77, CC79]. Sample has been used to implement many different static analyses [FFJ12, ZFC12, CFC11] and was extended with abstract semantics necessary for the analysis of TouchDevelop scripts. Recently, improved collection support [Bon13] has been added. TouchBoost was also used to implement a cost estimation for scripts [FSB14]. An overview of the TouchBoost architecture is shown in Figure 1. The front-end TouchDevelop compiler takes local scripts or JSON ASTs from the TouchDevelop cloud and translates them into the intermediate representation `Simple`. Afterwards, the forward abstract interpreter analyzes the programs. The analysis is instantiated with a composition of different abstract domains and computes the abstract program semantics. In the end, TouchBoost produces an error report with the analysis results.

TouchDevelop is an attractive target for static analysis, since an open web API provides access to all published scripts and their development histories. At the time of writing, already more than 90 000 scripts have been published.

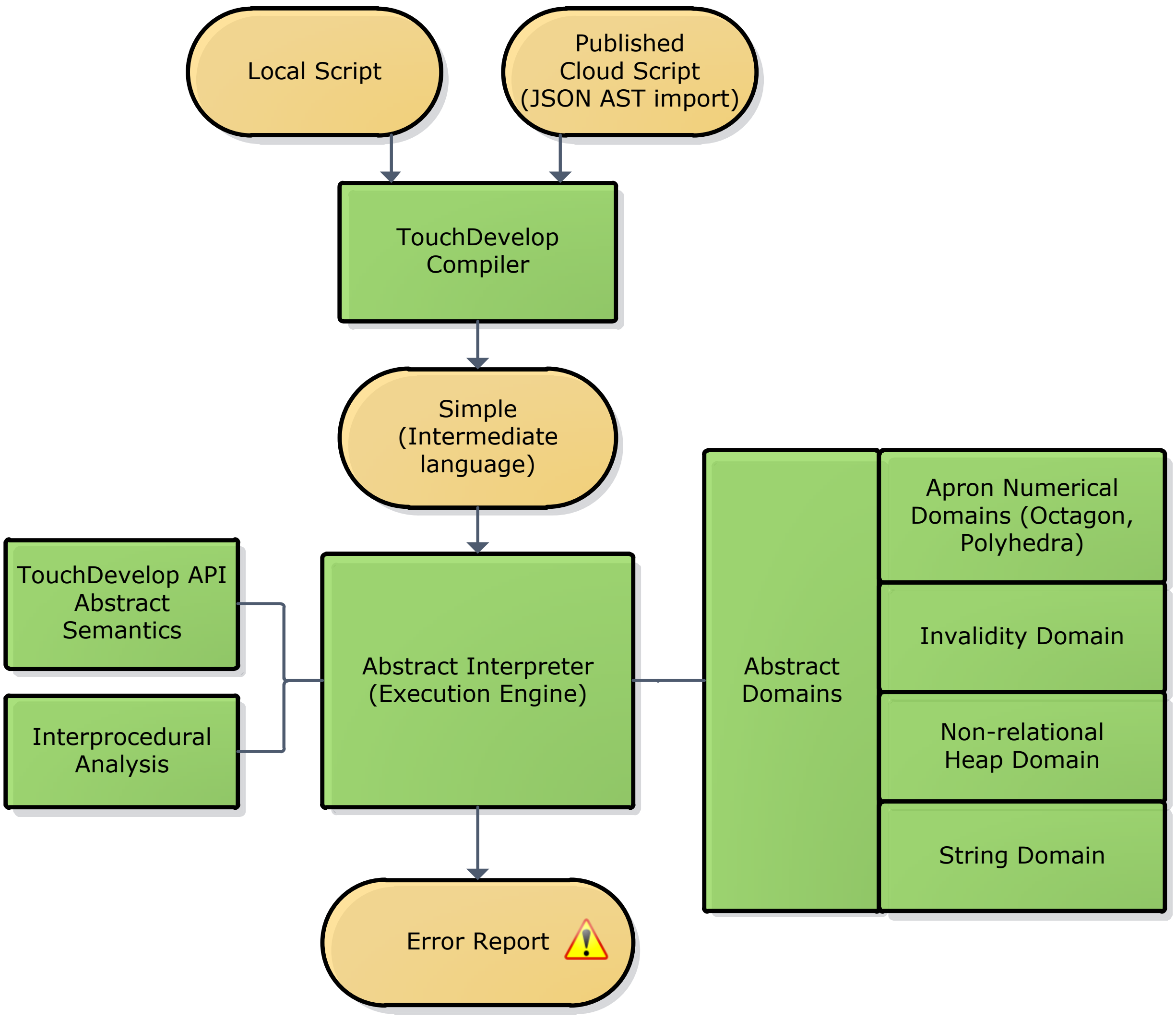

Local Script
Published Cloud Script (JSON AST import)
TouchDevelop Compiler
Simple (Intermediate language)
TouchDevelop API Abstract Semantics
Interprocedural Analysis
Abstract Interpreter (Execution Engine)
Abstract Domains
Apron Numerical Domains (Octagon, Polyhedra)
Invalidity Domain
Non-relational Heap Domain
String Domain
Error Report

Figure 1: The architecture of TouchBoost

3 Foundations

In this section we explain the foundations of the technique we use, focusing on the general concepts and ignoring the specifics for TouchBoost which will be illustrated later.

Sample is based on the *abstract interpretation* theory [CC77, CC79, Cou78], making static and sound reasoning about programs possible. Like standard abstract interpreters, it computes a set of program states that a run of the program may encounter when started from a given initial state. This is done by tracking static information along the program flow and is referred to as *forward abstract interpretation*.

However, as mentioned in 1.1, we need a static analysis that is able to reason *backwards* from an error towards the program entry. There is well established theory [Cou78, Riv05] that describes a dual *backward abstract interpretation* to find states that may reach a given final state. Both abstract interpretations can also be combined into a more precise variant called *refining interpretation* which uses both forward and backward information and that we decided to adopt in this thesis.

We now describe more formally the framework of abstract interpretation and then show how we make use of a refining abstract interpreter.

3.1 Forward Abstract Interpretation

Before defining a static analysis and its approximations, the formal meaning of underlying programs must be specified. At the lowest level, the operational semantics of a programming languages define what a given program does. They give rise to a transition system encoding the program behavior.

Transition systems. A transition system is a pair $(\Sigma, \rightarrow)$ where Σ is the set of states and $\rightarrow \in \mathcal{P}(S \times S)$ the transition relation. We write $\sigma \rightarrow \sigma'$ for $(\sigma, \sigma') \in \rightarrow$, if the system may transition from state σ to σ'. For our programs, Σ consists of all the possible program states which have some further structure and consist of variable environment and the program heap, i.e. $\Sigma = \mathit{Env} \times \mathit{Heap}$.

Reachability semantics. The *concrete forward reachability semantics* $\mathcal{C}_\mathcal{I} \in \mathcal{P}(\Sigma)$ of a program then describes all program states that may be reached from a set of possible initial states $\mathcal{I}$. The operator

$$\mathrm{post}(S) = \{\sigma \in \Sigma \mid \exists \sigma' \in S\colon\ \sigma' \rightarrow \sigma\} \tag{2}$$

yields all the direct successor states of set of pre-states. Note that a single state σ can only ever have more than once successor if the program is non-deterministic. Using this operator, we can express the forward semantics as a least fixed point (the order being subset inclusion)

$$\mathcal{C}_\mathcal{I} = \mathrm{lfp}\, \lambda X.\, \mathcal{I} \cup \mathrm{post}(X) \tag{3}$$

$\mathcal{C}_\mathcal{I}$ also has a convenient, isomorphic representation which collects all the states that may occur at each point the program code, essentially describing local invariants. Given a function $extractPP : \Sigma \to ProgramPoint$ that extracts the program point of a state, we can express these invariants in terms of $\mathcal{C}_{\mathcal{I}}$:

$$\begin{aligned} \hat{\mathcal{C}}_{\mathcal{I}} &: ProgramPoint \to \mathcal{P}(\Sigma) \\ \hat{\mathcal{C}}_{\mathcal{I}}(pp) &\mapsto \{\sigma \in \mathcal{C}_{\mathcal{I}} \mid extractPP(\sigma) = pp\} \end{aligned} \tag{4}$$

Programs as equation systems. However, program semantics expressed directly using transition systems as in Equation 3 are not used in practice, since such semantics are monolithic and not built from reusable components [Min12]. Instead, one prefers an easier formulation as the (least) solution of an equation system instead. In this equation system, we describe how the states at different program points are related to each other with the help of *concrete transfer functions* $\overrightarrow{c}[\![s]\!] : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$. For every statement s, $\overrightarrow{c}[\![s]\!](\sigma_{pre})$ returns all the post-states the program can be in after executing the statement on any pre-state in σ_{pre}.

As an example, the factorial function in Figure 2 a) can be described by the following equation system in b).

```
action fact(n: Number)
 returns (r: Number) {
  (P1) r := 1;
  (P2) while n >= 1 do {
    (P3) r := r * n;
    (P4) n := n - 1;
    (P5)
  }
  (P6) contract->assert(r >= 1);
}
```

(a) Program code

$$\begin{aligned}
\hat{\mathcal{C}}_{\mathcal{I}}(P1) &= \mathcal{I} \\
\hat{\mathcal{C}}_{\mathcal{I}}(P2) &= \overrightarrow{c}[\![r := 1]\!](\hat{\mathcal{C}}_{\mathcal{I}}(P1)) \cup \hat{\mathcal{C}}_{\mathcal{I}}(P5) \\
\hat{\mathcal{C}}_{\mathcal{I}}(P3) &= \overrightarrow{c}[\![n >= 1]\!](\hat{\mathcal{C}}_{\mathcal{I}}(P2)) \\
\hat{\mathcal{C}}_{\mathcal{I}}(P4) &= \overrightarrow{c}[\![r := r * n]\!](\hat{\mathcal{C}}_{\mathcal{I}}(P3)) \\
\hat{\mathcal{C}}_{\mathcal{I}}(P5) &= \overrightarrow{c}[\![n := n - 1]\!](\hat{\mathcal{C}}_{\mathcal{I}}(P4)) \\
\hat{\mathcal{C}}_{\mathcal{I}}(P6) &= \overrightarrow{c}[\![n < 1]\!](\hat{\mathcal{C}}_{\mathcal{I}}(P2))
\end{aligned}$$

(b) Equation system defining concrete (collecting) semantics

Figure 2: Factorial computation

Abstract Semantics. So far we only formalized the concrete semantics of programs. However, computations of concrete fixed points are not feasible. Programs can have an infinite number of executions and many interesting properties of programs are undecidable in general. We must therefore abstract from the concrete semantics and compute a sound over-approximation of the program states. Abstract interpretation provides a comprehensive framework to construct such sound abstractions.

The *concrete domain* is the lattice $(D = \mathcal{P}(\Sigma), \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ that represents concrete program states. The corresponding *abstract domain* is the lattice $(D^\sharp, \sqsubseteq^\sharp, \sqcup^\sharp, \sqcap^\sharp, \bot^\sharp, \top^\sharp)$ holding all the abstract states approximating the concrete ones. The two domains are related via a *galois connection* $D \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} D^\sharp$, where the *abstraction function* $\alpha : D \to D^\sharp$

approximates a concrete element and the *concretization function* $\gamma : D^\sharp \to D$ returns the possible concrete states represented by an abstract one.

Furthermore, for each concrete transfer function $\overrightarrow{c}[\![s]\!]$ for a statement s, an *abstract transfer function*

$$\overrightarrow{f}[\![s]\!] : D^\sharp \to D^\sharp \tag{5}$$

needs to be provided. Since this function needs to over-approximate the concrete one, all states produced by $\overrightarrow{c}[\![s]\!]$ applied to an arbitrary pre-state σ_{pre} must also be included in the abstract result. This is expressed by the soundness condition:

$$\forall \sigma_{pre} \in D^\sharp : \overrightarrow{c}[\![s]\!](\gamma(\sigma_{pre})) \sqsubseteq \gamma(\overrightarrow{f}[\![s]\!](\sigma_{pre})) \tag{6}$$

The main task when designing abstract domains is thus coming up with efficient yet precise and sound transfer functions. The abstract forward semantics $\hat{\mathcal{F}} : ProgramPoint \to D^\sharp$ of a complete program can then be expressed as an equation system in the same manner as for the concrete semantics. For our example in Figure 2 this would be:

$$\begin{aligned}
\hat{\mathcal{F}}_\mathcal{I}(P1) &= \mathcal{I} \\
\hat{\mathcal{F}}_\mathcal{I}(P2) &= \overrightarrow{f}[\![r := 1]\!](\hat{\mathcal{F}}_\mathcal{I}(P1)) \ \cup \ \hat{\mathcal{F}}_\mathcal{I}(P5) \\
\hat{\mathcal{F}}_\mathcal{I}(P3) &= \overrightarrow{f}[\![n >= 1]\!](\hat{\mathcal{F}}_\mathcal{I}(P2)) \\
\hat{\mathcal{F}}_\mathcal{I}(P4) &= \overrightarrow{f}[\![r := r * n]\!](\hat{\mathcal{F}}_\mathcal{I}(P3)) \\
\hat{\mathcal{F}}_\mathcal{I}(P5) &= \overrightarrow{f}[\![n := n - 1]\!](\hat{\mathcal{F}}_\mathcal{I}(P4)) \\
\hat{\mathcal{F}}_\mathcal{I}(P6) &= \overrightarrow{f}[\![n < 1]\!](\hat{\mathcal{F}}_\mathcal{I}(P2))
\end{aligned} \tag{7}$$

Abstract Interpreter. Equation systems like in Equation 7 can be solved iteratively by an abstract interpreter. Initially, let $\hat{\mathcal{F}}^1_\mathcal{I} = \lambda p.\bot$. If we then view the right hand sides of the equations as rules to update the corresponding entries in $\hat{\mathcal{F}}_\mathcal{I}$, we get a series of iterates $\hat{\mathcal{F}}^1_\mathcal{I} = \lambda p.\bot, \ \hat{\mathcal{F}}^2_\mathcal{I}, \ \cdots$. For example, we have

$$\hat{\mathcal{F}}^{k+1}_\mathcal{I}(P5) = \overrightarrow{f}[\![n := n - 1]\!](\hat{\mathcal{F}}^k_\mathcal{I}(P4))$$

If we continue this iteration, we eventually reach a fixed point. It is well known that the same result is obtained in the end whether one updates all the entries simultaneously, or individually by a so-called *chaotic iteration* [Bou93] which allows many different iteration schemes.

In Sample we use a work list based algorithm to perform this iteration on our programs represented in a traditional control flow graph (CFG) which consists of blocks of statements that are connected with conditional edges.

Pseudo code for the iteration is displayed in Algorithm 1. The algorithm proceeds as follows: We initialize the work list with the entry CFG block and the first state of the entry block with $\mathcal{I}$. Until the work list is empty, we keep removing CFG blocks from the list. For each retrieved list entry *block*, we have to recompute its current entry state. Since control may transfer from any predecessor block to the current one, we have to join all the predecessor states, but we can additionally assume the edge condition for each incoming edge. This is expressed by the operation $\dot{\bigsqcup}$ in the pseudo-code. If the resulting abstract updated entry state is smaller ($\sqsubseteq^\sharp$) than the block's old entry state $\hat{\mathcal{F}}_\mathcal{I}(block.entry)$, the iteration has not stabilized yet and we need to recompute the semantics of the block on the newer state. Updating the forward semantics of a block is straightforward: The forward transfer function $\overrightarrow{f}[\![stmt]\!]$ for every statement *stmt* in the block is evaluated in sequence. Since successor blocks may depend on the updated exit state of the current block and may be influenced, we add all of them to the work list and continue processing blocks.

Note the iteration may fail to converge within a reasonable number of steps. This is due to abstract domains with lattices of infinite height or program structures that are expensive to analyze such as nested loops. We detect slow convergence by counting the number of times a block is scheduled for recomputation (*slowConvergenceAt* in Algorithm 1). When a threshold is exceeded, we apply a widening operation that makes the abstract state larger to ensure convergence.

Algorithm 1 Forward abstract interpreter: CFG iteration

```
1:  function ForwardInterpret(cfg)
2:      $\hat{\mathcal{F}}_\mathcal{I}(p) \leftarrow \begin{cases} \mathcal{I} & \text{if } p = cfg.entry \\ \bot & \text{otherwise} \end{cases}$
3:      $worklist \leftarrow \{entryBlock\}$
4:      while $worklist \neq \varnothing$ do
5:          $block \leftarrow chooseNext(worklist)$
6:          $worklist \leftarrow worklist - \{block\}$
7:          $blockEntry \leftarrow \dot{\bigsqcup} predecessorExits(block)$
8:          $oldBlockEntry \leftarrow \hat{\mathcal{F}}_\mathcal{I}(block.entry)$
9:          if $blockEntry \not\sqsubseteq^\sharp oldBlockEntry$ then
10:             if $slowConvergenceAt(blockEntry)$ then
11:                 $blockEntry \leftarrow widen(oldBlockEntry, blockEntry)$
12:             end if
13:             $currentState \leftarrow blockEntry$
14:             for $statement \leftarrow block.statements$ do
15:                 $\hat{\mathcal{F}}_\mathcal{I}(statement.programPoint) \leftarrow currentState$
16:                 $currentState \leftarrow \overrightarrow{f}[\![statement]\!](currentState)$
17:             end for
18:             $\hat{\mathcal{F}}_\mathcal{I}(block.exit) \leftarrow currentState$
19:             $worklist \leftarrow worklist \cup successors(block)$
20:         end if
21:     end while
22: end function
```

3.2 Backward Abstract Interpretation

With an established formalism for the forward analysis, we can now state the details for backward abstract interpretation. The goal is now to reason backwards from a set of end states $\mathcal{E}$ (usually containing erroneous states), and to arrive at a set of concrete states at the program entry that may potentially lead to the ones in $\mathcal{E}$.

Backward reachability semantics. Analogously to the forward case, we describe the concrete backward semantics with the help of the transition system, but this time with an operator returning the predecessors of state sets:

$$\text{pre}(S) = \{\sigma \in \Sigma \mid \exists \sigma' \in S\colon\ \sigma \to \sigma'\} \tag{8}$$

$$\overleftarrow{\mathcal{C}}_{\mathcal{E}} = \text{lfp}\, \lambda X.\, \mathcal{E} \cup \text{pre}(X) \tag{9}$$

Abstract semantic equations and transfer functions. To express the abstract backward program semantics $\hat{\mathcal{B}}_{\mathcal{E}} : \mathit{ProgramPoint} \to D^\sharp$ similar to $\hat{\mathcal{F}}_{\mathcal{I}}$ as the solution of an equation sytem, we need to define *abstract backward transfer functions*

$$\overleftarrow{b}\,[\![s]\!] : D^\sharp \to D^\sharp \tag{10}$$

for all corresponding forward transfer functions $\overrightarrow{f}\,[\![s]\!]$. $\overleftarrow{b}\,[\![s]\!](\sigma_{post})$ must over-approximate all the concrete pre-states the program could execute from when it reaches one of the post-states in σ_{post} directly by executing statement s. The formal soundness condition is:

$$\forall \sigma_{post} \in D^\sharp : \overrightarrow{c}[\![s]\!]^{-1}\,[\gamma(\sigma_{post})] \sqsubseteq \gamma(\overleftarrow{b}\,[\![s]\!](\sigma_{post})) \tag{11}$$

where we use $h^{-1}[A]$ to denote the preimage of A under h. The semantic equation system for our example now is:

$$\begin{aligned}
\hat{\mathcal{B}}_{\mathcal{E}}(P1) &= \overleftarrow{b}\,[\![r := 1]\!](\hat{\mathcal{B}}_{\mathcal{E}}(P2)\\
\hat{\mathcal{B}}_{\mathcal{E}}(P2) &= \overleftarrow{b}\,[\![n >= 1]\!](\hat{\mathcal{B}}_{\mathcal{E}}(P3)) \cup \overleftarrow{b}\,[\![n < 1]\!](\hat{\mathcal{B}}_{\mathcal{E}}(P6))\\
\hat{\mathcal{B}}_{\mathcal{E}}(P3) &= \overleftarrow{b}\,[\![r := r * n]\!](\hat{\mathcal{B}}_{\mathcal{E}}(P4))\\
\hat{\mathcal{B}}_{\mathcal{E}}(P4) &= \overleftarrow{b}\,[\![n := n - 1]\!](\hat{\mathcal{B}}_{\mathcal{E}}(P5))\\
\hat{\mathcal{B}}_{\mathcal{E}}(P5) &= \hat{\mathcal{B}}_{\mathcal{E}}(P2)\\
\hat{\mathcal{B}}_{\mathcal{E}}(P6) &= \mathcal{E}
\end{aligned}$$

Backward interpreter. To find an approximative fixed point of this equation system, we can again apply a work list iteration. As the procedure is mostly dual to the forward case in Algorithm 1, we do not reproduce the pseudo-code here. Basically, instead of starting with the entry CFG block, we initialize the work list with the block containing the final state. Until the work list is empty, a CFG block is selected. We then check if its old exit state is general enough by joining all the block entry states of the successors and comparing the result to the old one. If not, we update the block by going through the statements in the block in reverse order, executing the backward transfer function $\overleftarrow{b}\,[\![stmt]\!]$ on each of them. Widening is applied as in the forward iteration.

Relationship between forward and backward semantics. As a theoretical side remark, we note that the concrete forward and backward semantics of programs are completely dual to each other. In fact, the `post` operator can be expressed in terms of `pre` and so we would not need the additional notation in our formalism. Furthermore, the classical weakest precondition and strongest postconditions semantics of programs can be shown to be equivalent [Cou97]. However, as pointed out by [Cou98], we obtain different results once abstraction is introduced and the semantics is approximated. In practice, this means that one nevertheless needs to define separate forward and backward transfer functions for all statements.

3.3 Refining Backward Interpretation

Forward-backward combination. If we want to express the concrete states that are reached for executions that both start from a set of initial states $\mathcal{I}$ and end up in a set of final states $\mathcal{E}$, we may take the intersection of the two fixed points of the concrete forward and backward reachability semantics:

$$\mathcal{C}_{\mathcal{I}} \cap \overleftarrow{\mathcal{C}_{\mathcal{E}}} = (\operatorname{lfp} \lambda X.\, \mathcal{I} \cup \operatorname{post}(X)) \cap (\operatorname{lfp} \lambda X.\, \mathcal{E} \cup \operatorname{pre}(X)) \tag{12}$$

Naïve abstract semantics. In the abstract, it is safe to do the same and perform a *separate* forward and backward abstract interpretation as described in sections 3.1 and 3.2. The forward-backward semantics is then the meet of the two results:

$$\lambda l\ .\ \hat{\mathcal{F}}_{\mathcal{I}}(l)\ \sqcap^{\sharp}\ \hat{\mathcal{B}}_{\mathcal{E}}(l) \tag{13}$$

Refining abstract semantics. However, a naïve intersection of forward and backward analysis results as in Equation 13 can be very imprecise. The precision can be improved if we resort to a mixture between forward and backward analysis which can reuse the information obtained by the forward abstract interpretation *during* the backward interpretation. In the literature, this is known as the *refining forward-backward process* [Riv05, CC92, Cou78].

The forward interpretation is applied as described earlier and gives us $\hat{\mathcal{F}}_{\mathcal{I}}$. However, for the subsequent backward interpretation, we modify the abstract transfer functions to also take the pre-condition into account that was inferred to hold before the statement and which is available in $\hat{\mathcal{F}}_{\mathcal{I}}$. We call those new transformers the *refining backward* *transfer functions*:

$$\overleftarrow{b^{ref}}[\![s]\!] : D^\sharp \times D^\sharp \to D^\sharp \tag{14}$$

$$\overleftarrow{b^{ref}}[\![s]\!](\sigma_{oldPre}, \sigma_{post}) \mapsto \sigma_{refinedPre} \tag{15}$$

Such a transfer function for statement s maps the current post-state σ_{post} together with the pre-state from the forward analysis σ_{oldPre} to a new, *refined* pre-state $\sigma_{refinedPre}$.

To see the difference between $\overleftarrow{b}[\![s]\!]$ and $\overleftarrow{b^{ref}}[\![s]\!]$ consider the statement

(ℓ_1) `x := x + 1` (ℓ_2)

that is executed at program label ℓ_1 before reaching ℓ_2. Assume the backward interpreter starts with an abstract post-state σ_{post} at ℓ_2 for which holds $2 \leq x \leq 3$. $\overleftarrow{b}[\![x := x + 1]\!]$ would then produce a pre-state containing $1 \leq x \leq 2$. On the other hand, if it is already know from the forward analysis that $0 \leq x \leq 1$ must hold at ℓ_1, the refining transfer function can take this pre-state $\hat{\mathcal{F}}_{\mathcal{I}}(\ell_1) = \sigma_{oldPre}$ and compute a more precise *refined* pre-state

$$\sigma_{refinedPre} = \overleftarrow{b^{ref}}[\![x := x + 1]\!](\sigma_{oldPre}, \sigma_{post})$$

for which we know exactly $x = 2$.

Note that a refining transfer function is free to take a simple intersection (meet) between the backward result and the pre-state as in the naïve forward-backward combination above (Equation 13). For example, we could define the refining semantics of *stmt* to be:

$$\overleftarrow{b^{ref}}[\![stmt]\!](\sigma_{oldPre}, \sigma_{post}) = \overleftarrow{b}[\![stmt]\!](\sigma_{post}) \sqcap^\sharp \sigma_{oldPre}$$

This simple fallback can be applied for some operations, while other statements can take advantage of σ_{oldPre} in more sophisticated ways.

Soundness. The modified soundness condition for refining transfer functions $\overleftarrow{b^{ref}}[\![s]\!]$ reads

$$\forall \sigma_{post}, \sigma_{oldPre} \in D^\sharp : \overrightarrow{c}[\![s]\!]^{-1}[\gamma(\sigma_{post})] \sqcap \gamma(\sigma_{oldPre}) \sqsubseteq \gamma(\overleftarrow{b^{ref}}[\![s]\!](\sigma_{oldPre}, \sigma_{post})) \tag{16}$$

Refining backward interpreter. Pseudo-code for a refining backward interpreter is shown in Algorithm 2. It is a blend of the forward and backward interpreters discussed in Section 3.1 and Section 3.2, respectively. As in the backward case, it also processes CFG blocks in reverse order and propagates abstract states from block entries to their predecessor exits. The key difference however is that the interpreter also has access to $\hat{\mathcal{F}}_{\mathcal{I}}$ and uses these pre-states when computing the refining transfer functions for block statements. Also note that the interpretation does not have to proceed from the CFG exit, but it may start with the end states $\mathcal{E}$ from an arbitrary statement (program point *endLocation*) in the CFG.

Algorithm 2 Refining backward interpreter: CFG iteration

```
1: function RefiningBackwardInterpret(cfg, $\hat{\mathcal{F}}_{\mathcal{I}}, \mathcal{E}, endLocation$)
2:     $\forall p : \hat{\mathcal{B}}^{ref}_{\mathcal{E}}(p) \leftarrow \bot$
3:     $worklist \leftarrow \{endLocation.block\}$
4:     while $worklist \neq \varnothing$ do
5:         $block \leftarrow chooseNext(worklist)$
6:         $worklist \leftarrow worklist - \{block\}$
7:         $blockExit \leftarrow \dot{\bigsqcup} successorEntries(block)$
8:         $oldBlockExit \leftarrow \hat{\mathcal{B}}^{ref}_{\mathcal{E}}(block.exitPoint)$
9:         if $blockExit \not\sqsubseteq^{\sharp} oldBlockExit$ then
10:            Widen state when necessary
11:            if $slowConvergenceAt(block)$ then
12:                $blockExit \leftarrow widen(oldBlockExit, blockExit)$
13:            end if
14:            $currentState \leftarrow blockExit$
15:            for $statement \leftarrow reverse(block.statements)$ do
16:                $\hat{\mathcal{B}}^{ref}_{\mathcal{E}}(statement.programPoint) \leftarrow currentState$
17:                $oldPreState \leftarrow \hat{\mathcal{F}}_{\mathcal{I}}(statement.programPoint)$
18:                $currentState \leftarrow \overleftarrow{b^{ref}}[\![statement]\!](oldPreState,\ currentState)$
19:                if $statement.location = endLocation$ then
20:                    $currentState \leftarrow currentState \sqcup \mathcal{E}$
21:                end if
22:            end for
23:            $\hat{\mathcal{B}}^{ref}_{\mathcal{E}}(block.entry) \leftarrow currentState$
24:            $worklist \leftarrow worklist \cup predecessors(block)$
25:        end if
26:    end while
27: end function
```

3.4 Illustration of Approach

We now illustrate how we intend infer definite counterexamples with the help of the new refining backward analysis.

The example code in Listing 2 serves in a step-by-step explanation. It takes two integers and performs some arithmetic computations, including a division. A division by zero may occur because nothing restricts the input parameters x and y. The program is annotated with the invariants inferred by the forward analysis, displayed in blue. Note that these formulas are always of a very restricted form, limited by the expressiveness of the used abstract domains. In our example, we used the octagon numerical domain, so all invariants are conjunctions with terms of the form $\pm var_1 \pm var_2 = const$. This is also the reason why we get a generic $\top$ state after the assignment in line 3, as we cannot represent the fact $t = x - y - 1$ with octagon constraints.

```
1 action main(x: Number, y: Number) {
2   (s) { ⊤ }
3   t := x - y - 1;
4   { ⊤ }
5   if (x >= y) {
6     { y − x <= 0 }
7     (lbl) r := 1 / t;
8   }
9 }
```

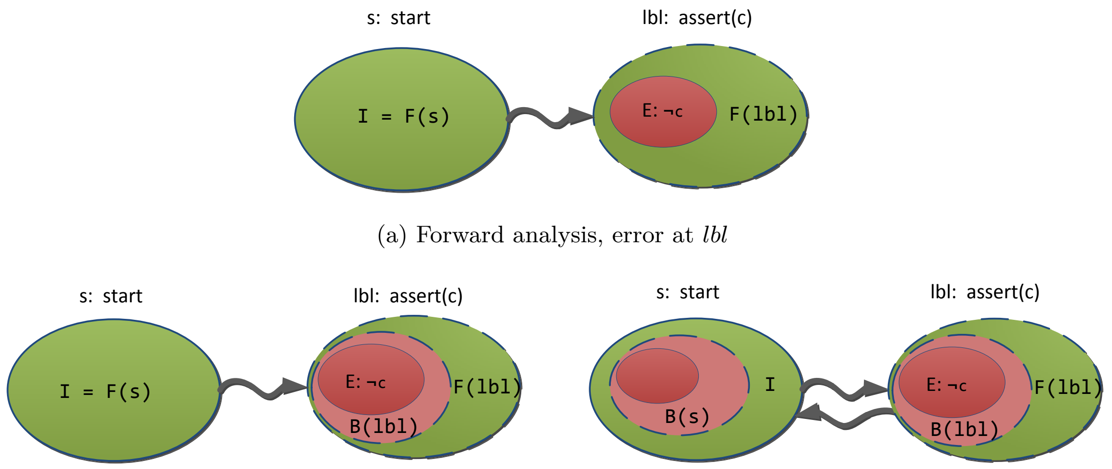
Listing 2: Example annotated with forward analysis results

The forward analysis registers an *abstract error* at line 7 because the implicitly generated assertion $t \neq 0$ before the division cannot be shown to hold in the abstract semantics, i.e.

$$\{y - x \leq 0\} \ \sqcap^\sharp \ \{t = 0\} \ \neq \ \bot$$

We therefore start the investigation of this abstract error using the backward analysis, beginning at *lbl* with an over-approximation of the error states $\hat{\mathcal{B}}^{ref}_{\mathcal{E}}(lbl) = \{y - x \leq 0\} \ \sqcap^\sharp \ \{t = 0\} = \mathcal{E}$

Just as the forward analysis over-approximates the program states that can be actually reached from a set of inputs, our refining backward analysis seeks to find an over-approximation of all the states that lead to this error state. At the same time, it *refines* the invariants obtained by the forward analysis. This makes sure that everything we infer about the states leading to an error is consistent with the forward analysis. At the entry label s, we finally end up with a refined entry state that contains all program states where $y - x = -1$, which is precisely the condition for the error to happen. The program annotated in red with the computed backward invariants is shown in Listing 3.



(a) Forward analysis, error at *lbl*

(b) Over-approximated error state

(c) Result of backward analysis

Figure 3: Visualization of approach. *Green*: Over-approximation of forward-reachable states , *Red*: Error states, *Light red*: Over-approximation of states leading to error, *Dashed*: Over-approximations

```
action main(x: Number, y: Number) {
  (s) { ⊤ } { y − x = −1 }
  t := x - y - 1;
  { ⊤ } { y − x <= 0, t = 0 }
  if (x >= y) {
    { y − x <= 0 }{ y − x <= 0, t = 0 }
    (lbl) r := 1 / t;
  }
}
```

Listing 3: Example annotated with forward- and backward invariants

Figure 3 visualizes this procedure for a violated assertion $assert(c)$ such as division by zero check in our example. The sets of all possible states at both the program entry s and the assertion label lbl are drawn. The forward analysis in (a) determines that some concrete states E where c does not hold may be reached at lbl – the green and red sets intersect. Since the concrete error states at lbl may not always be precisely expressed by the abstract domain, the backward analysis starts with an over-approximation $\hat{\mathcal{B}}^{ref}_{\mathcal{E}}(lbl) = \mathcal{E}^{\sharp}$ of them (Figure 3 b). The refining backward analysis then results in a refined entry state at s which is always a subset of all the possible original entry states $\mathcal{I}$ (Figure 3 c).

In our example, it was possible to refine the original entry state without any constraints on the inputs into an abstract entry state $\hat{\mathcal{B}}^{ref}_{\mathcal{E}}(s)$ that contains *precisely* the x and y parameter values leading to the error. However, since we use over-approximations, there are programs for which the concretization $\gamma(\hat{\mathcal{B}}^{ref}_{\mathcal{E}}(s))$ may also contain states that do not entail an error. Therefore, we need to pick concretized potential counterexamples and run the program with the inputs. If the execution leads to the error, we can be sure that the counterexample is not spurious. While we apply standard *concrete testing*, more sophisticated techniques could be used.

For some abstract errors, it may also happen that the refined entry state becomes $\perp$. In this case, we conclude that the forward analysis produced a *false alarm*, because there do not possibly exist any concrete inputs that could cause the error, i.e. $\gamma(\perp) = \varnothing$. The causes for false alarms are manifold but always boil down to over-approximations that the abstract program semantics makes. Abstract interpreters are designed to be *sound*: They do not to miss any possible program behavior, but at the same time they are almost never *complete*, i.e. they lose information when calculating the fixed points of the collecting semantics. Often, the loss of information occurs because the abstract transfer functions are not implemented precisely enough. Another very common reason for false alarms is that the abstract domains themselves are simply not sufficiently expressive.

Finally, if none of the cases above occur, we have to give up and declare the alarm as *undecided* (inconclusive), since we could neither present a counterexample nor declare it false. Algorithm 3 summarizes our error investigation process for the set of all alarms $\mathcal{A}$ produced by the forward interpreter.

Algorithm 3 Abstract error investigation process, pseudo-code.

```
function ANALYZE(m: Method)
    // Compute forward semantics, collect alarms.
    $(\mathcal{F}, \mathcal{A}) \leftarrow forwardInterpret(m)$
    // Investigate alarms
    for all $a \in \mathcal{A}$ do
        // Compute abstract error state
        $\mathcal{E} \leftarrow \mathcal{F}(a.programpoint) \sqcap^{\sharp} \{ \neg\, a.assertion \}$
        $\mathcal{B}^{ref} \leftarrow backwardInterpret(m, \mathcal{F}, \mathcal{E}, a.programpoint)$
        $\sigma_{refinedEntry} \leftarrow \mathcal{B}^{ref}(m.entry)$
        if $\sigma_{refinedEntry} = \perp$ then
            Report a as false alarm
        else
            // Randomly test candidates
            for $i \leftarrow 1$ to #tries do
                // Generate a concrete entry state
                $\sigma_{concreteEntry} \leftarrow \gamma_{next}(\sigma_{refinedEntry})$
                $errorReached \leftarrow concreteRun(s, \sigma_{concreteEntry}, a)$
                if $errorReached$ then
                    Report a as actual error, give counterexample $\sigma_{concreteEntry}$
                end if
            end for
            Report alarm a as undecided
        end if
    end for
end function
```

4 Backward Analysis in TouchBoost

The implementation of a refining backward analysis in TouchBoost required a number of changes that we now describe. They mainly concern the extension of the abstract domain with refining backward transfer functions for operations such as variable assignment. We also show how we handle challenging analysis aspects like interprocedural semantics and reasoning about non-deterministic code.

4.1 The TouchBoost Abstract Domain

4.1.1 Domain Structure

The foundations above refer to a generic lattice $D^\sharp$ of abstract values. In TouchBoost, this abstract domain is not monolithic. Instead, it is a composition of multiple specialized domains where each domain abstracts particular aspects of concrete runtime values. This makes the analysis parametric and different domains may be instantiated if required. The composition mechanism is usually the cartesian product.

At the highest level, a state consists of two subdomains: A *heap domain* that abstracts the concrete heap structure of the program, and a second *semantic domain* that abstracts the values of identifiers.

The semantic domain is further decomposed into separate domains that track the different value types of identifiers: The *invalid domain* abstracts the validity (non-nullness) of identifiers. Another domain tracks the values of string identifiers. Finally, the *numerical domain* abstracts only the concrete values of all numerical identifiers.

The heap domain on the other hand consists of two separate domains: One for variable identifiers in the environment and another one for heap identifiers in the program store.

To sum up, the structure of TouchBoost's abstract domain expressed in terms of cartesian product compositions boils down to:

$$\begin{aligned} TouchBoostDomain &= D^\sharp = SemanticDomain \times HeapDomain \\ SemanticDomain &= StringDomain \times (InvalidDomain \times NumericalDomain) \\ HeapDomain &= VariableEnv \times HeapEnv \end{aligned}$$

4.1.2 Backward Semantics for Product Domains

Each analyzed TouchDevelop statement causes a series of operations to be issued on the abstract state. For example, an assignment to a variable results in the *assign* operation. Other common operations include the creation of objects and the assumption of conditions.

For all these operations, we need to implement a (refining) backward transfer function $\overleftarrow{b^{ref}}$. Because the TouchBoost abstract domain is structured into cartesian products, the transfer functions of subdomains can be applied individually. For example, the semantics of $assign(x, y)$ which assigns y to x is expressed as follows. Given a full TouchBoost pre-state $\sigma = (\sigma_S, \sigma_H)$ and post-state $\sigma' = (\sigma'_S, \sigma'_H)$ which consist of the semantic and heap domains, we can write the full transfer function in terms of the subdomain transfer functions:

$$\overleftarrow{b^{ref}}[\![assign(x, y)]\!](\sigma, \sigma') = \left(\overleftarrow{b^{ref}}[\![assign(x, y)]\!]_S(\sigma_S, \sigma'_S), \quad \overleftarrow{b^{ref}}[\![assign(x, y)]\!]_H(\sigma_H, \sigma'_H)\right)$$

Hence, in the following description we show how the different abstract domains support the backward operations.

4.2 Semantic Domain

The following essential operations must be implemented for all the semantic sub-domains:

- $createVariable(v)$ introduces a new identifier v whose runtime should be abstracted in the domain
- $removeVariable(v)$ removes a given identifier v from the domain.
- $assume(expr)$ causes the semantic domain to assume that the boolean-valued symbolic expression $expr$ holds for all concrete values in the current abstract state. This enables the computation of a "smaller", more precise abstraction. Assumptions are particularly important for control flow constructs. For example in the analysis of "`if (cond) then ... else ...`" we perform $assume(cond)$ and $assume(\neg cond)$ when entering the true and the false branch, respectively.
- $assign(id, expr)$ assigns the contents of expression $expr$ to the given identifier. The expression can for example be a constant value, another identifier or a more complex symbolic term with arithmetic or boolean operators.

We first discuss the handling of the operations that is common to all semantic subdomains. The backward semantics for $assign$ needs specific implementations, so they are described separately for each the numeric, the invalid and the string domain.

Variable creation and removal. In theory, these two operations simply have the opposite effect to their forward versions, when executed backward:

$$\overleftarrow{b^{ref}}[\![createVariable(v)]\!] = \overrightarrow{f}[\![removeVariable(v)]\!] \tag{17}$$

$$\overleftarrow{b^{ref}}[\![removeVariable(v)]\!] = \overrightarrow{f}[\![createVariable(v)]\!] \tag{18}$$

However, the creation of variable identifiers is handled rather loosely in Sample: When $createVariable$ is invoked, it only creates a new variable when there does not already exist one with the same name. This property is used by TouchBoost for the semantics of assignments `var := expr` to a variable. $createVariable(var)$ is always first executed to make sure the variable exists. We therefore decided to omit an explicit removal of the variable *var* when executing backwards. We leave the task of getting rid of a variable that does not exist in the pre-state but in the post-state to a modified greatest lower bound operation $\sqcap$ described in Section 4.4.

Assuming expressions. As mentioned above, the most common usage of $assume(expr)$ is the assumption of branching conditions. For the backward semantics, we are given a post-state at the entry of a branch and are interested in the pre-states that may reach our post-state. In a forward execution the branch would only have been taken if the branch condition *expr* was true, so we may do the same as in the forward semantics and assume the condition:

$$\overleftarrow{b^{ref}}[\![assume(expr)]\!](\sigma_{oldPre}, \sigma_{post}) = \overrightarrow{f}[\![assume(expr)]\!](\sigma_{post}) \tag{19}$$

4.2.1 Numerical Domain

The numerical domain tracks the values of numerical identifiers. Each domain state maintains a set of known numerical identifiers and describes an over-approximation of all the concrete values these identifiers may take on. It also manages boolean identifiers whose truth values are encoded with the numbers 0 and 1.

In TouchBoost, different numerical domain implementations can be selected by the user. The *interval domain* abstracts each identifier value with lower and upper bounds. It is non-relational and cannot capture relations such the equality of identifiers. The more precise *octagon domain* describes relationships between variables with constraints of the form $\pm var_1 \pm var_2 = const$. Yet more expressive is the polyhedra domain, which discovers general linear inequalities among identifiers. The octagon domain offers more performance in exchange for precision and is therefore the default choice for analyses.

Let Num from now on denote the selected numerical domain.

Backward assignment. At first, it may seem that all a backward assignment can do is to forget the current value in the post-state of the variable that is assigned. This is illustrated in the very simple example in Figure 4 (a) where we want to compute the backward semantics of the assignment `n := x` on a post-state with $n = 0$. By forgetting the value of n, we lose all information and obtain a pre-state of $\top$. This is certainly sound, but very imprecise.

Even though the value of the assigned variable is forgotten because it was erased by the assignment, we can learn more about the state before the assignment. In our example, we know that $n = x$ must hold after the statement. Since we already have $n = 0$ and the assignment does not change x, we can infer that before the assignment $x = 0$ holds (Figure 4 b). This simple relational reasoning can be delegated to the numerical domain Num in the form of an assumption $assume(n = x)$. However, this argument is not valid in general when the assigned identifier is also referenced in the right-hand side, such as in $n := n + 1$.

The following observation generalizes the idea above: A backward assignment transformer for `x := expr` can be thought to perform a substitution that replaces all occurences of the identifier x in the symbolic representation of the post-state with the expression *expr*, i.e. the pre-state becomes $\sigma_{post}[expr/x]$. This idea directly corresponds to the assignment axiom in Hoare logic. However, the constraints of our abstract domains are not arbitrary expressions, but part of a limited fragment of formulas, such as linear inequalities, so the result of an exact substitution may not be expressible in the abstract domain. We therefore require the numerical domain Num to provide an operation $subst_{Num}(\sigma, id, expr)$ which approximates a substitution that replaces an identifier *id* with an expression *expr* in the abstract state σ. We assume the substitution to be sound, that is, we have

$$\alpha\left(\overrightarrow{c}[\![assign(id, expr)]\!]^{-1}(\gamma(\sigma))\right) \sqsubseteq^\sharp subst_{Num}(\sigma, id, expr)$$

We can then define the refining backward transfer function for assignments as

$$\overleftarrow{b}^{ref}[\![assign(id, expr)]\!](\sigma_{oldPre}, \sigma_{post}) = subst_{Num}(\sigma_{post}, id, expr) \sqcap^\sharp \sigma_{oldPre}$$

Imprecise generic substitution. With the existing functionality of the numerical domains, a simple generic substitution $subst_{generic}(\sigma, id, expr)$ can be defined for all domains Num:

1. Create a fresh identifier id' that represents the "old" value of *id* before the assignment. Nothing is known about it yet.
2. Enable the domain to establish a relationship between id' and *id* by executing the assumption $assume(id = expr[id'/id])$, where all occurences of *id* in *expr* are syntactically replaced with id'. For example, we would perform $assume(n = n'+1)$ for the substitution $subst_{generic}(\sigma, n, n+1)$ which is used to handle an assignment $n := n + 1$.
3. Remove the identifier *id* that refers to the value in the post state.
4. Rename id' to *id*.

$\{\ \top\ \}$
`n := x;`
$\{\ n = 0\ \}$

(a) Naive transformer: Forgetting assigned identifier value

$\{\ x = 0\ \}$
`n := x;`
$\{\ n = 0\ \}$

(b) Improvement: Substitution in abstract state

Figure 4: Backward assignment

Substitution using Apron. In our actual implementation of the backward assignment, we do not use the generic substitution just presented. Instead, we rely on a more precise variant offered directly by the Apron library [JM09]. Our domains already make heavy use of Apron's fast native code by calling an extensive but low-level Java API provided by the `japron` Java binding.

Apron's substitution functionality is able to take into account the internals of the particular numerical domain and also takes advantage of the old pre-state to further refine the result, going much further than taking a greatest lower bound. Complex non-linear expressions are linearized [Min06] to get a symbolic over-approximation, which is then used to make the substitution more precise.

Discussion. Consider Figure 5 for an example where the preciser Apron substitution with linearization improves the precision somewhat compared to a naive generic substitution. The code computes the square of a variable x, for which we initially know $1 \leq x \leq 4$. The forward and backward invariants are again illustrated in blue and red, respectively. We use the strict polyhedra domain to perform a backward analysis where an exact computation of the entry state would require taking the square root of x. Non-linear constraints are beyond the expressiveness of the domain but the situation can be mitigated by the linearization of Apron. Note that in (a) the imprecise substitution leads to an entry state of $1 \leq x \leq 4$, which is the same as the forward invariant already known, i.e. the backward analysis did not infer anything useful. On the other hand, the better Apron substitution is able to narrow down the initial states that may lead to the assertion violation and yields $3.75 \leq x \leq 4$, which is much closer to the exact (concrete) solution $3.87... \leq x \leq 4$. A similar precision improvement is also illustrated in [Riv05] where they use the interval domain on a different example.

```
// Exact states leading to
    error:
// {3.87.. ≤ x ≤ 4}

{ 1 ≤ x ≤ 4 }{ 1 ≤ x ≤ 4 }
x := x * x;
{ 1 ≤ x ≤ 16 }{ 15 > x ≤ 16 }
contract->assert(x <= 15,
  "may not hold");
```

(a) Generic substitution, glb with pre-state

```
// Exact states leading to
    error:
// {3.87.. ≤ x ≤ 4}

{ 1 ≤ x ≤ 4 }{ 3.75 ≤ x ≤ 4 }
x := x * x;
{ 1 ≤ x ≤ 16 }{ 15 < x ≤ 16 }
contract->assert(x <= 15,
  "may not hold");
```

(b) Apron: Substitution with linearization

Figure 5: Precision comparison of different backward assignments

4.2.2 Invalid Domain

All types in TouchDevelop contain a special *invalid* value which corresponds to *null* in other languages. It is typically used for uninitialized variables and as a return value of unsuccessful operations. Because we want to prevent dereferences of this invalid value, the *Invalid* abstract domain in TouchBoost separately tracks the validity of identifiers.

$$\begin{tikzcd}[column sep=small] & \{valid, invalid\} \arrow[dl, dash] \arrow[dr, dash] & \\ \{valid\} \arrow[dr, dash] & & \{invalid\} \arrow[dl, dash] \\ & \bot = \varnothing & \end{tikzcd}$$

Figure 6: $ValidityValue$ lattice

All TouchDevelop values are abstracted by the $ValidityValue$ domain which determines whether a value is definitely *invalid*, definitely *valid* or unknown. Its lattice is a simple powerset lattice of with the set $\{valid, invalid\}$, displayed in Figure 6. The domain that abstracts the validity of all identifier values has the structure of a *functional domain*: All its abstract values are maps from identifiers to elements of $ValidityValue$. Formally:

$$\begin{aligned} ValidityValue &= (\mathcal{P}(\{Valid, Invalid\}),\ \subseteq,\ \cup,\ \cap, \varnothing,\ \{Valid, Invalid\}) \\ InvalidDomain &= \quad Identifier \to \mathcal{P}(\{Valid, Invalid\} \end{aligned} \tag{20}$$

Backward Assignment. The $InvalidDomain$ is non-relational as the individual map entries abstract the values of identifiers separately. As a consequence, the backward assignment is simple and can be implemented by forgetting the validity value of the identifier that is being assigned, with a prior assumption the equality between the identifier and assigned expression:

$$\begin{aligned} &\overleftarrow{b^{ref}}[\![assign(id, expr)]\!](vmap_{oldPre}, vmap_{post}) = \\ &\left(\quad \lambda k.\ \begin{cases} \{valid, invalid\} & \text{if } k = id \\ \overrightarrow{f}[\![assume(id = expr)]\!](vmap_{post})(k) & \text{otherwise} \end{cases} \right) \sqcap vmap_{oldPre} \end{aligned} \tag{21}$$

Example: When executing backward the sequence of assignments

```
y := x; z := y
```

from a post-state $vmap \in InvalidDomain$ where we know $vmap(z) = \{invalid\}$, we obtain a refined pre-state $vmap'$ with $vmap'(x) = \{invalid\}$. The initial validity values of both y and z are unknown since they are overwritten by the assignments. However, our equality assumption back-propagates the fact that z is invalid in the end to its origin x.

Detection of False Alarms. To be able to detect false alarms, we need the greatest lower bound $\sqcap$ of two $InvalidDomain$ states to become $\bot$ when the entries for an identifier contradict each other. One usage of the greatest lower bound occurs e.g. in the backward assignment defined above.

For example, assume that a backward operation results in state $vmap$ with $vmap(x) = \{invalid\}$ at a location where forward analysis already inferred the invariant $vmap'(x) =$

$\{valid\}$. Clearly, these facts are not compatible so there cannot be any concrete execution. This may happen during the analysis of a false alarm and enables us to detect it.

However, the default implementation of $\sqcap$ for functional domains in Sample such as *InvalidDomain* does not result in a $\bot$ state but only a bottom validity value $\varnothing$ for the particular identifier entry. I.e., for our example we need that $vmap \sqcap vmap' = \bot$ but instead only $(vmap \sqcap vmap')(x) = \varnothing$ holds.

Hence, we defined a custom greatest lower bound operation $\sqcap^{ID}$ for *InvalidDomain* that includes a special case for such situations and otherwise reuses the generic $\sqcap$ of functional domains (see Section 4.4):

$$(m_1 \sqcap^{ID} m_2) = \begin{cases} \bot & \text{if } \exists\, id : \begin{array}{l}(m_1(id) = \{Valid\} \wedge m_2(id) = \{Invalid\}) \vee \\ (m_2(id) = \{Invalid\} \wedge m_2(id) = \{Valid\})\end{array} \\ m_1 \sqcap m_2 & \text{otherwise} \end{cases} \tag{22}$$

4.2.3 String Domain

StringDomain is responsible for abstracting the runtime values of string identifiers. It is a non-relational functional domain like *InvalidDomain*. We only equipped it with minimal backward transformers that are sound but imprecise.

The backward assignment forgets the value of the identifier that is being assigned.

4.3 Heap Domain

The abstract heap domain in TouchBoost maintains an abstraction of all the allocated heap locations and the values of references.

Heap Abstraction. For every concrete reference (memory location) $r \in Ref$, we have a corresponding abstract heap identifier $heapId = \alpha_{ref}(r)$ returned by the abstraction function

$$\alpha_{ref} : Ref \to HeapId \tag{23}$$

We use a program-point based abstraction that assigns the same heap identifier to all references of objects created at the same program location. The number of heap identifiers therefore always remains bounded, for example when creating new objects in a loop. We call an identifier which has more than one concrete counterpart a *summary identifier*.

The abstract heap consists of two components: The abstract environment and store. Both are functional abstract domains with the keys being identifiers and the values members of the set domain of heap identifiers. I.e., for each variable identifier and heap identifier, there is an entry in the environment or store, respectively, with an over-approximation of all heap identifier it points to. Formally:

$$\begin{aligned} Heap &= Env \times Store \\ HeapIdSet &= \mathcal{P}(HeapId) \\ Env &: VarId \rightarrow HeapIdSet \\ Store &: HeapId \rightarrow HeapIdSet \end{aligned}$$

Since all identifiers have separate entries, we cannot explicitly state relationships among them. For example, we cannot represent the fact that two local variables must reference the same heap object. The domain is therefore non-relational.

Backward Assignment. Assignment operations $assign(id, expr)$ that modify our abstract heap structure come in two forms with slight semantic differences:

- $assign(id,\ id2)$ makes the identifier id point to all identifiers the identifier $id2$ may point to.
- $assign(id,\ \{hid1,\ hid2, \cdots\})$ causes id to point to the all the identifiers in the given set. It is the case with a "constant" right-hand side, so to speak. The heap identifier set usually originated from the evaluation of more complex symbolic expression.

For both versions, depending on whether the left-hand side id is a variable identifier or a general heap identifier, either the environment or the store is affected by this operation. In the backward assignment transfer function, we have to forget the identifier that was assigned to but instead of setting it to $\top$, the entry from the old pre-state is used:

$$\overleftarrow{b^{ref}}[\![assign(varId,\ expr)]\!]((env_{oldPre}, store_{oldPre}), (env_{post}, store_{post})) = \\ \left(\left(\lambda\, k.\begin{cases} env_{oldPre}(varId) & \text{if } k = varId \\ env_{post}(k) & \text{otherwise} \end{cases}\right),\ store_{post}\right)$$

$$\overleftarrow{b^{ref}}[\![assign(heapId,\ expr)]\!]((env_{oldPre}, store_{post}), (env_{post}, store_{post})) = \\ \left(env_{post},\ \left(\lambda\, k.\begin{cases} store_{oldPre}(heapId) & \text{if } k = heapId \\ store_{post}(k) & \text{otherwise} \end{cases}\right)\right)$$

In the case of assignments with id sets $assign(id,\ \{hid,\ hid2,\ \cdots\})$, this is all we can do. On the other hand, for better precision in the case $assign(id, id2)$, we may first execute an $assume(id\ ==\ id2)$ to establish the equality of the identifiers after the assignment.

Example: Let us assume we have a post-state with two local variables p, r with two allocated objects represented by the heap identifiers $objId1$ and $objId2$. Further we know that p may point to both objects, but r only to the first, i.e. $env(p) = \{objId1, objId2\}$, $env(r) = \{objId1\}$. Computing the backward semantics of assignment $assign(r, p)$, we can infer a pre-state where we know p must point to $objId1$, since the assumption can take the intersection of both heap identifiers sets:

```
{ p → {objId1}, r → ⊤ }
r := p;
{ p → {objId1, objId2}, r → {objId1} }
```

Object Removal. The heap identifiers associated with a heap object, namely for the object itself and its fields, are created in the abstract state by $\overrightarrow{f}[\![createObject(typ, programPoint)]\!]$. Naturally, we remove them again in the backward version of this transformer.

4.4 Greatest Lower Bounds

Two aspects concerning the handling of identifiers in the greatest lower bound (meet) operation required special attention throughout all abstract domains.

Removal of Uncommon Identifiers. It is important that created identifiers are removed again when the program is interpreted backwards. However, new abstract identifiers are often implicitly introduced by TouchDevelop statements and the scope of identifiers is rather undisciplined in Sample. For example, the forward semantics of an assignment $x := y$ may implicitly create an identifier for x in case it does not exist yet in the pre-state.

As the greatest lower bound with the pre-state is taken by a refining backward transfer function, this operation provides a good opportunity to get rid of any superfluous identifiers not present in the pre-state. We thus made sure that the greatest lower bound always removes identifiers not common to both abstract states.

The greatest lower bound for the numerical domain was already implemented in this way. On the other hand, this is not the case for all functional domains (in particular, our $InvalidDomain$, $StringDomain$ and $HeapDomain$). As stated earlier, the abstract values of a functional domain are maps from identifiers to elements of another lattice

$$(ValueDomain, \sqsubseteq^V, \sqcup^V, \sqcap^V, \bot^V, \top^V). \tag{24}$$

The existing $\sqcap$ on the elements of functional domains is defined as:

$$\begin{aligned} dom(a \sqcap b) &= dom(a) \cup dom(b) \qquad (25) \\ (a \sqcap b)(id) &= \begin{cases} a(id) \sqcap^V b(id) & \text{if } id \in dom(a) \cap dom(b) \\ a(id) \sqcap^V default & \text{if } id \in dom(a) - dom(b) \\ b(id) \sqcap^V default & \text{if } id \in dom(b) - dom(a) \end{cases} \qquad (26) \end{aligned}$$

The value $default \in ValueDomain$ can be provided by specific functional domain implementations, making the operation more flexible but less uniform across domains. The problematic part of this definition is that the domain of the resulting map $dom(a \sqcap b)$ always contains all the identifiers, including the ones not common to a and b.

We therefore use a stricter stricter and simpler definition of the greatest lower bound $\sqcap^s$ for all functional domains in the backward analysis:

$$\begin{aligned} dom(a \sqcap^s b) &= dom(a) \cap dom(b) \qquad (27) \\ (a \sqcap^s b)(id) &= a(id) \sqcap^V b(id) \qquad (28) \end{aligned}$$

Unnecessary Summary Identifiers. An identifier may become a *summary identifier* during the forward analysis. As explained in Section 4.3, a summary identifier soundly abstracts one or more concrete objects (values). To guarantee sound updates, abstract operations involving summary identifiers have to be conservative and perform weak updates on the abstract values for these identifiers. Let us represent the property of identifiers of an abstract state being summary in a map

$$summaryIds : D^\sharp \to Identifier \to \{true, false\} \qquad (29)$$

Once an identifier an identifier id becomes a summary, it normally stays so in all abstract successor states computed from it. However, in the backward execution we want them to become normal, non-summary identifiers again if possible. This effect can be achieved in the greatest lower bound operation: If id is a summary in one state a (i.e., $summaryIds(a)(id) = true$) but not in the other state b, it can become non-summary in the result. The only way an identifier stays a summary after the operation is when it already is in both original states. That is, the update of the summary information can be expressed as:

$$summaryIds(a \sqcap b)(id) = summaryIds(a)(id) \wedge summaryIds(b)(id) \qquad (30)$$

4.5 Native Method Semantics

Sample's program representation called "Simple" aims to be as generic as possible and is restricted to the most essential elements common across imperative programming languages: The supported statements are constants, variable references, assignments,

```
CheckNonNegative(widthParam)
CheckNonNegative(heightParam)
New(TPicture.typ,Map(
  TPicture.field_width -> widthParam,
  TPicture.field_height -> heightParam
))(preState)
```

Listing 4: Native method semantics for `media->create picture(widthParam, heightParam)`

```
createObject(programPoint, typ)
assignField(programPoint, width, 10)
assignField(programPoint, height, 5)
assignField(programPoint, location, invalid)
...
```

Listing 5: Operations on abstract state during call to `media->create picture(10, 5)`

field accesses and method calls. However, it allows almost arbitrary nesting of these constructs and does not enforce a particular structure such as static single assignment form (SSA).

Any language statement that is not directly translatable to one of the mentioned constructs is modelled as a method call, including calls to the API of the language libraries. While the abstract semantics of the pre-defined elements like assignments is implemented generically, all method call semantics must be modelled manually with so-called *native method semantics*.

The TouchDevelop API comprises more than 100 types with the number of members (properties) exceeding 1500, and it is steadily growing. All of these plus a few custom helper methods for language constructs are specified via native method semantics. To make the specifications more succinct, TouchBoost offers expressive Scala helper methods that can be composed to act like a embedded domain specific language (EDSL). For example, a call `media->create picture(width, height)` to create a picture can be (slightly simplified) specified using Scala code like in Listing 4.

These forward native semantics are already implemented in TouchBoost and basically give us an abstract transfer function $\overrightarrow{f}[\![nm]\!]$ for each native method nm of the API. We now need the corresponding $\overleftarrow{b^{ref}}[\![nm]\!]$ backward transfer functions for the backward analysis as well. Given the sheer number of such methods, it would be an enormous task to specify them all by hand. However, at closer look the rich semantics in TouchBoost are expressed in terms of lower-level operations on states. Consider for example some of the state operations that are invoked for the call `media->create picture(10, 5)`, displayed in Listing 5.

```
eh   :=   σ_pre basicOp | eh ; eh | σ_pre if (cond) eh else eh

basicOp   :=   assignVariable | createObject | ...
```

Listing 6: Structure of execution history

We have backward transformers for all such lower-level operations like assignments. It thus natural to define the backward semantics of a native method as the reverse applications of the backward transformers of all the elementary operations which the method entails. In the current TouchBoost architecture, these operations are not part of the AST/CFG of our programs and are only invoked implicitly when executing the DSL-like rich semantic helpers. The intermediate states which we need to supply as forward pre-states to the *refining* backward transfer functions are not saved anywhere either.

To work around this architectural limitation with reasonable effort, we decided to wrap our abstract state and record all forward operations and their effect during forward execution as part of our state. These new *history states* maintain their own execution history in addition to the normal abstract state:

$$HistoryState = D^\sharp \times ExecutionHistory \tag{31}$$

The execution history is a sequence of forward operations but may also branch due to conditional semantic definitions. Loops are not possible, as the semantic helpers do not express general control flow. We also make sure to save all the intermediate forward states σ_{pre} to be used in the backward run. Listing 6 shows the structure of the execution histories. After the forward history has been recorded, we can backward interpret it starting from the given post-state.

Note that the described workaround is only needed during the abstract interpretation of statements with native method call. It is not necessary to keep the history of operations across multiple statements when interpreting the control flow graph.

4.6 Support for Non-Determinism

Another challenging feature of the backward analysis is reasoning about non-deterministic program inputs. We first explain the nature of non-determinism in TouchDevelop and then show how we support it in our analysis.

4.6.1 Non-Determinism in TouchDevelop

Many language constructs and the execution model of TouchDevelop scripts are inherently non-deterministic. Basically everything that is not pre-determined before the execution of a script may be considered as non-determinism. In particular, non-deterministic behavior occurs for the following reasons:

Random number generation. Scripts may consume pseudo-random numbers as produced by library calls like `Math->random`. This is e.g. often used by small game engines for "jump-and-run" games.

Direct user inputs. Users can be prompted with a dialogue to input numbers, strings and make boolean "yes-no" choices. Apart from statistical properties, these inputs are indistinguishable from randomly generated input.

Interactions with environment. Every interaction with the environment can be seen as a form of non-determinism. Many TouchDevelop API calls require accessing external resources. For example, a script may issue a HTTP request to a host, and the result (success or failure of request, content of response) is not fixed ahead of time.

Event execution order. As the execution model of TouchDevelop is event-based, we have no concurrent interleaving of execution threads. However, the order in which events are scheduled and executed is not defined.

To see why a proper treatment of non-determinism matters, consider the very simple example in Listing 7. For the possible division by zero in line 5, the backward analysis infers the entry state to be $\top$, since the user may or may not enter zero and we need a valid over-approximation. There is no way to narrow down the input state because we cannot refer to the non-deterministic input *before* the assignment to n.

```
1 action main()
2   { ⊤ }
3   var n := wall->ask number("positive number?")
4   { n = 0 }
5   var result := 1 / n
```

Listing 7: Simple case of non-deterministic input

4.6.2 Approach

Internal vs. External Non-Determinism. Non-deterministic decisions can be thought to be made by an external entity, outside of our program. This is obviously the case for all interactions with the network. But also other forms of non-determinism may also be interpreted like that. On the other hand, one may have the viewpoint that the system has already predetermined all non-deterministic choices itself, consulting some internal hidden state whenever decisions are to be made.

Non-determinism source identifiers. We decided to handle non-determinism (excluding the form of event execution order) by converting the external determinism to an internal one, so that we can reason about it as part of the state, entirely within our framework. For all program points where a non-deterministic value is produced, we add internal state recording that value. These values are associated with a new kind of identifier $NonDetId$ which represents a *source of non-determinism.* Such identifiers possess a name, type and a program point describing where the non-deterministic access occurs. The program point information is later used to look up the relevant identifier in the state:

$$\begin{aligned} &NonDetId &&\subseteq Identifier && (32)\\ &name: && NonDetId \to String && (33)\\ &type: && NonDetId \to Type && (34)\\ &pp: && NonDetId \to ProgramPoint && (35) \end{aligned}$$

The identifiers are added to the initial analysis state and tracked throughout the forward and backward analysis. To make use of the new identifiers, we have to change the abstract semantics of all the methods returning a non-deterministic result. So far, such methods always returned a constant expression representing a $\top$ value of the required type. For example, a non-deterministic method with a boolean result would return the symbolic expression $\{true, false\}$. Since we want to reason about the values of non-deterministic decisions, we override this behavior and return our new identifiers instead.

For the example in Listing 7, we would introduce a new identifier *nid* for the number returned by the call to `wall->ask number`, e.g. with $name(nid) =$ nondet_pp3_16, $type(nid) = Number$, $pp(nid) =$ "line 3 column 16". Then we simply return *nid* as the resulting expression. Later during the backward analysis of the division-by-zero error, we can establish the fact $n = nid = 0$ and end up with $nid = 0$ in the initial state.

Additional assumptions. Some TouchDevelop API methods are modelled with forward semantics that do more than just return a $\top$ value. For example, `Math->random(limit)` produces pseudo-random integers between 0 and *limit* (excluding the upper bound). Its old forward semantics returned a symbolic expression $0 \xrightarrow{to} limit$ representing such a number range. Because we want to return a non-deterministic identifier, we have to establish these additional constraints for the identifier. We do this by making a series of additional assumptions on the state before returning the resulting identifier. For the `Math->random` case returning the identifier *nid*, this would be

$$assume(nid = (0 \xrightarrow{to} limit))$$

Summary Identifiers. The same non-determinism source may be accessed more than once during program execution, but our forward semantics always returns the same non-deterministic identifier. If we do not treat this situation differently from single accesses, we run into an unsoundness. Listing 8 displays a constructed example where this happens: The user is asked two times for a number, and the program asserts that they are the same. Obviously, the assertion has to fail because there is nothing requiring the inputs to be equal, but if we always return the same identifier, the checks succeeds. The reason is that our identifier actually stands for more than one concrete value instance.

In Sample, we already have a mechanism to deal with cases where an abstract identifier represents multiple concrete identifiers, which is used in the context of heap-allocated values. As mentioned in Section 4.3, multiple objects created at the same program point are approximated by the same identifier, which we call a *summary identifier*. Here, we have essentially the same situation where we want a program-point bounded abstraction to get a finite over-approximation.

Summary identifiers are treated specially in the abstract domains. For assignments, this means *weak updates* to the abstract state instead of strong ones. Assuming expressions is handled similarly. This prevents unsound under-approximations, often by paying with precision.

```
action main()
  var first = 0;
  var second = 0;
  for 0 < i <= 2 do {
    var n := wall->ask number(”next number?”)
    if (i = 1) then {
      first := n;
    } else {
      second := n;
    }
  }
  assert(first = second);
```

Listing 8: Non-determinism source accessed multiple times: Summary identifier necessary to prevent unsoundness.

The task now is to determine which sources of non-determinism need to have a summary identifier, i.e. to provide the information:

$$summary : \quad NonDetId \rightarrow \{true, false\} \tag{36}$$

so that the identifiers can be created accordingly in the initial state. While we could syntactically scan the program for method calls nested inside loops which exhibit non-determinism, this would be unreliable and awkward. Listing 9 shows that we also have to take into account the interprocedural behavior of our scripts starting from the entry action. Even though the access on line 9 is not directly in a loop, it may be called multiple times from main.

```
1  action main() {
2    var x := wall->ask number(”next number?”)
3    for 0 < i <= 2 do {
4      var y := sub();
5    }
6  }
7
8  private action sub() returns (r: Number) {
9    r := wall->ask number(”next number?”)
10 }
```

Listing 9: Counting calls to non-deterministic methods

Instead, we decided to instrument the existing infrastructure and implement a simple data flow analysis which is run before the main analysis. Its task is to collect all the non-deterministic accesses and determine conservatively whether they may happen multiple times at the same location.

Our state consists of the *access map domain* which is the functional domain

$$(AccessMapDomain,\ \sqsubseteq^{AM},\ \sqcup^{AM},\ \sqcap^{AM},\ \bot^{AM},\ \top^{AM}) \tag{37}$$

with the keys being the non-deterministic identifiers and the values access counts:

$$\begin{aligned} AccessMapDomain :&\quad NonDetId \to AccessCount & (38)\\ AccessCount \ =&\ \{NoAccess,\ SingleAccess,\ MultipleAccess\} & (39)\end{aligned}$$

AccessCount forms the lattice $(AccessCount, \sqsubseteq^{AC}, \sqcup^{AC}, \sqcap^{AC}, \bot^{AC}, \top^{AC})$ where the element order is a linear chain and the according least upper bound and greatest lower bound are defined naturally:

$$\bot^{AC} = NoAccess \sqsubseteq^{AC} SingleAccess \sqsubseteq^{AC} MultipleAccess = \top^{AC} \tag{40}$$

All forward semantics of native methods that access a source of non-determinism make sure to call the operation $accessNondet(id)$ on the current state. We then define its forward transfer function for the *AccessMapDomain* to be

$$\begin{aligned} &\overrightarrow{f}[\![accessNondet(id)]\!](accessMap) = \\ &\qquad \lambda k.\ \begin{cases} SingleAccess & \text{if } k = id \land accessMap(id) = NoAccess \\ MultipleAccess & \text{if } k = id \land accessMap(id) = SingleAccess \\ accessMap(k) & \text{otherwise} \end{cases}\end{aligned}$$

The join $\sqcup^{AM}$ and meet $\sqcap^{AM}$ of *AccessMapDomain* are defined in the usual manner:

$$\begin{aligned} \sqcup^{AM}(a,b) &= \lambda k.\ a(k)\ \sqcup^{AC}\ b(k) \\ \sqcap^{AM}(a,b) &= \lambda k.\ a(k)\ \sqcap^{AC}\ b(k) \end{aligned}$$

Precision Improvement for Loops. While the introduction of summary identifiers is necessary, it hurts the precision of our backward analysis since we cannot infer any useful facts about them. As a counter-measure, we implemented a syntactic unrolling of loops. This way, we get non-deterministic non-summary identifiers for the first k unrolled loop iterations in addition to the summary identifier for the remaining iterations. The backward analysis can then establish more precise constraints on these identifiers for the states that lead to an error.

4.7 Interprocedural Analysis

In the examples presented so far, we constrained ourselves to TouchDevelop scripts with a single method. However, most useful real-world scripts consist of more than one method.

Units of execution in scripts are divided into *actions* and *events*: Actions are normal methods while events are event handlers that are executed in response to some runtime event like the user touching the screen or shaking the phone. Both are chunks of code taking parameters and possibly returning values, and we collectively refer to them as *methods* when the distinction does not matter. Actions can be made public or private, which enables some visibility control and encapsulation.

4.7.1 Concrete and Abstract Execution Model

The *concrete execution model* of scripts is a simple event loop. One of the public actions (by default `main`) is executed first. Once it terminates, all triggered events are handled. Both events and the actions may call other actions in turn, and after their completion the remaining events are dealt with. A script may be run multiple times and may keep persistent state between invocations.

TouchBoost provides support for the interprocedural analysis of scripts. There are several aspects to this: The analyzer needs to know what to do with the abstract state when a method calls another one. It should also reuse computed semantics modularly to guarantee reasonable performance. Moreover, it must soundly handle the possibility that a method calls itself in the case of recursion. Finally, it should take into account the behavior of all events that may be executed.

Figure 7 (inspired by a similar figure in [Bon13]) shows an overview of the *abstract execution model* that TouchBoost uses for the interprocedural analysis The execution could start with any public action, so we execute them all “in parallel” on the initial state and join the results. After that, the same happens with all the events. But since more than one event can be executed in sequence, we have to make sure to take the least fixed point (or an over-approximation in the case of widening) of this repeated event execution. We also compute the fixed point of the repeated application of this whole process so multiple script executions are considered.

4.7.2 Method Summaries and Forward Method Call Semantics

To describe the extensions for the backward analysis, we first need to explain how the abstract semantics of user-defined methods are handled.

For every method, TouchBoost keeps a *method summary* which is a description of the forward semantics that this method may exhibit in the context of the given program. We maintain these summaries in a map, slightly adopting our previous notation for the forward states $\mathcal{F}$ so that the states are now associated explicitly with a method:

$$\mathit{Summary}\mathcal{F} : \mathit{Method} \to \mathit{ProgramPoint} \to D^\sharp$$

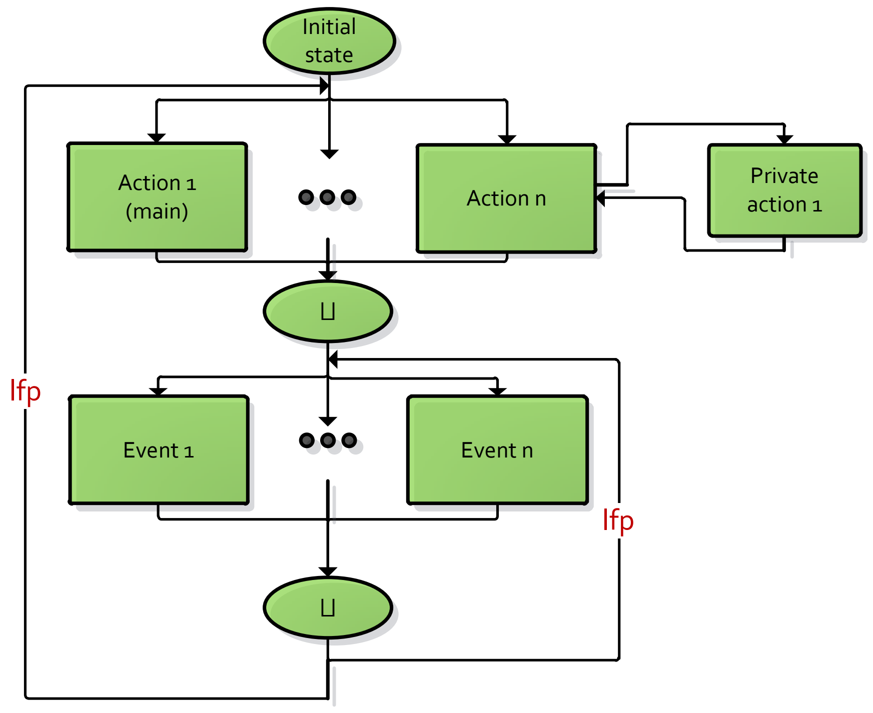


Figure 7: Interprocedural analysis: Abstract execution model

E.g., we denote the entry state in the method summary of `main` by

$$Summary\mathcal{F}(\texttt{main})(entryPP)$$

where $entryPP$ is the program point of the first method statement.

In TouchDevelop, methods may call others by accessing them as properties of the global singleton `code`. For instance, `r := code->m(a1, a2)` calls method m with two arguments and assigns the returned result.

The forward semantics $\overrightarrow{f}[\![code\rightarrow method(arg1, arg2, \cdots]\!]$ first has to enter the method and establish the relationship between the passed actual method arguments with the formal ones. It then consults the summary for the method. If the state on which the method is called is contained in the summary entry state, we can directly reuse the abstract summary exit state. If not, the summary is not yet general enough: there may exist additional exit states and we need to reanalyze the method's body with the forward interpreter on the new input. So while a method may be called more than once, the behavior of all these calls is described by the same summary. Before returning from the call, the analysis exits from the method scope by removing the locally scoped variables and makes a return value from the output parameter identifiers.

Special care needs to be taken for directly or indirectly recursive methods. The interprocedural analysis maintains a set that over-approximates all the methods that may be on the call stack at a given time. When a method being called is already contained in this set, we detect a possible recursion and introduce additional approximation to ensure both soundness and termination. A simple but very imprecise way is to return $\top$ in such a case. TouchBoost implements a more sophisticated recursion handling that we omit here.

Algorithm 4 contains informal pseudo-code for what we have just described.

Algorithm 4 Informal interprocedural call semantics, forward analysis.

```
function $\overrightarrow{f}[\![code\to method(concreteArgs)]\!](\sigma_{pre}, methodStack)$
    // Enter scope, create local variables for arguments and outputs,
    // assign concrete arguments
    $\sigma' \leftarrow \overrightarrow{f}[\![enterMethod(method, concreteArgs]\!](\sigma_{pre})$
    if $method \in methodStack$ then
        // Possibly recursive call detected (naive strategy)
        $\sigma'' \leftarrow \top$
    else
        $s \leftarrow summary\mathcal{F}(method)$
        if $\sigma' \sqsubseteq s(entry)$ then
            // Reuse summary
            $\sigma'' := s(exit)$
        else
            // Generalize summary, interpret method body again
            $methodStack \leftarrow methodStack \cup \{method\}$
            $s' \leftarrow forwardInterpret(method.cfg, s, postEnterState)$
            $summmary\mathcal{F}(method) \leftarrow s'$
            $methodStack \leftarrow methodStack - \{method\}$
            $\sigma'' := s'(exit)$
        end if
    end if
    // Exit scope, purge local state except return variables
    $\sigma_{post} \leftarrow \overrightarrow{f}[\![exitMethod(method)]\!](sigma'')$
    return $\sigma_{post}$
end function
```

4.7.3 Backward Summaries

The approach of the forward analysis can be used during the backward analysis as well. In addition to the forward summaries, we also keep *backward summaries*

$$Summary\mathcal{B} : Method \to ProgramPoint \to D^\sharp$$

which save the states that possibly reach the abstract error currently under investigation. The abstract semantics of calls to user-defined methods $\overleftarrow{b^{ref}}[\![code\to method(arg1, arg2, \cdots]\!]$

can be implemented similarly to Algorithm 4, but this time using the refining interpreter on the method body. The forward summaries are reused for the refinement and remain unchanged while the backward summaries as generalized when necessary. Algorithm 5 contains informal code for this process.

The two operations for entering and exiting a method (like setting up a stack frame) have their backward transfer functions implemented as follows:

- $\overleftarrow{b^{ref}}[\![exitMethod(method)]\!]$: We recreate all local variables that are present in the forward pre-state but not in the backward post-state. This includes variables for the formal method parameters of the method, output parameters and possibly local variables that are assigned somewhere in the method. Furthermore, we undo the heap pruning of unreachable objects.
- $\overleftarrow{b^{ref}}[\![enterMethod(method)]\!]$: Undo the assignment of the parameter values and remove all formal method parameter variables.

Algorithm 5 Informal interprocedural call semantics, backward analysis.

```
function $\overleftarrow{b^{ref}}[\![code \to method(concreteArgs)]\!](\sigma_{pre}, \sigma_{post}, methodStack)$
    // Reuse forward summary
    $fs \leftarrow summary\mathcal{F}(method)$
    // Undo exiting from method scope
    $\sigma' \leftarrow \overleftarrow{b^{ref}}[\![exitMethod(method)]\!](fs(exit), \sigma_{post})$
    if $method \in methodStack$ then
        // Possibly recursive call detected (naive strategy)
        $\sigma'' \leftarrow \top$
    else
        $bs \leftarrow summary\mathcal{B}(method)$
        if $\sigma' \sqsubseteq bs(exit)$ then
            // Reuse summary
            $\sigma'' := bs(exit)$
        else
            // Generalize summary, backward interpret method body again
            $methodStack \leftarrow methodStack \cup \{method\}$
            $bs' \leftarrow refiningBackwardInterpret(method.cfg, fs, bs, \sigma')$
            $summmary\mathcal{B}(method) \leftarrow bs'$
            $methodStack \leftarrow methodStack - \{method\}$
            $\sigma'' := bs'(exit)$
        end if
    end if
    // Undo entering method scope
    $\sigma_{refinedPre} \leftarrow \overleftarrow{b^{ref}}[\![enterMethod(method, concreteArgs)]\!](\sigma_{pre}, \sigma'')$
    return $\sigma_{refinedPre}$
end function
```

4.7.4 Interprocedural Error Investigation Strategies

When we looked at isolated methods, the goal of the analysis was always obvious, namely to compute a refined entry state *at the start of the method that contains the alarm* and to try to synthesize a counterexample from it that leads to the error.

Backward analysis to top-level method entry. One might want to go further and try to determine from where and under what conditions the method with the alarm was called, and provide a counterexample starting from this point.

This proves to be difficult in general since in a full-scale interprocedural execution of a script, there are multiple locations in the program from which we could try to find a counterexample: While TouchDevelop scripts are typically started from `main`, they can have multiple entry actions. Furthermore, when events are executed before the alarm, a counterexample must include a sequence of all events executed up to that point. Considering all possibilities quickly leads to a combinatorial explosion.

Instead of tackling the problem in its full generality, we support a few simple scenarios that seem common enough.

Listing 10 shows an example where a private action `calc` is called by the top-level public action `main`. An alarm is produced in the private action, but clearly, the cause for it is a user input in its caller. If we search for a counterexample input for action `calc`, we learn that *param* must be non-positive. However, looking at a broader context, we could also determine that the parameter *fixed* in action `main` must be false and the non-deterministic user input a non-positive number. We see that in such a case, a non-local counterexample is helpful when trying to understand the problematic code.

However, the method with an alarm may be reached from multiple top-level methods. We therefore compute the call graph of a TouchBoost script and detect any call roots (top-level methods) from which the problematic method could be called. For each such call root, the backward analysis is executed assuming the program starts its executions from that method.

Note that this procedure is a heuristic since it considers only certain execution patterns and not all possible program behaviors anymore.

```
action main(fixed: Boolean) {
   if (not fixed) then {
      n := wall->ask number(”enter number?”);
      code->calc(n);
   } else {
      code->calc(1234);
   }
}

private action calc(param: Number) {
   // ...
   contract->assert(param > 0, ””);
```

```
  // ...
}
```

Listing 10: Alarm where non-local counterexample is helpful

Interprocedural False Alarm Detection. We also implemented experimental support for the interprocedural detection of false alarms, in particular when an alarm is reported in an event.

The idea is to issue a backward analysis run that propagates the error states all the way back to the start of the program. This can be done analogously to the forward abstract execution model in Figure 7, basically inverting all the arrows and computing the backward semantics starting from method exits instead of forward semantics from the entries.

Since before the event with the alarm another event may have been executed, we have to take a join over the backward analysis results of all possible events, and take the fixed point of this process since several events can be executed in sequence. After that, we propagate the resulting refined state back into all available public actions and take the join. The resulting state represents an over-approximation of all the initial program states that may potentially lead to the abstract error, considering *all* interprocedural execution behaviors. If we obtain $\bot$, we can be sure that no sequence of events of events causes the reported error.

A real-world use case of this technique is discussed later in Section 6.3.3

4.8 Collections

Collections are a central feature of TouchDevelop. TouchBoost has very recent support for tracking both may- and must-information about collection contents [Bon13]. Due to the complexity of the current implementation and time constraints, we were forced to implement coarse backward semantics and take some shortcuts.

Operations on abstract states that modify collections such as *insertCollectionElement* are handled by forgetting the collection contents. For the domain that keeps the may-contain over-approximation, the result is a collection that possibly contains anything. For the must-contain domain, a $\top$-valued collection identifier means we cannot guarantee the presence of any elements. Other operations that do not modify the collection such as for testing whether a collection contains a given element do not need an explicit backward version. One consequence of our imprecision is for example that we cannot infer entry states which lead to an error because some collection does not contain a particular element.

On the other hand, we can reason about the collection size since it is handled as an identifier in the numerical domain. E.g., if an error occurs because an invalid value is used after being returned from an empty collection, we can infer entry states with collection size zero.

5 Counterexample Generation

After running the backward analysis for a given abstract error, we can start the search for possible counterexamples. The inferred entry state is then ideally much more precise than the original unconstrained one, but not all concrete states represented by it have to lead to the error since we over-approximate the program behavior.

Our investigation (illustrated in Algorithm 3) proceeds by repeatedly picking concrete prospective states from the refined entry state and testing whether they result in the investigated error. The selection of concrete states may be guided by random decisions and heuristics. Clearly, we now only cover a subset of all concrete program traces, and may fail in general to obtain a conclusive classification of the alarm as true or false. If not, we can decide to simply drop the warning and not bother the user any further.

Thus, two mechanisms are needed: Firstly, a way to *concretize* abstract entry states and generate potential counterexamples. Secondly, a *testing procedure* to exercise the program on these inputs.

5.1 Concretization of Entry States

Abstract states can represent arbitrary many concrete ones – often even an infinite number. To enumerate them, we can exploit the well-defined structure of our abstract states which, as described in Section 4.1, consist of several sub-domains composed via cartesian product domains. Furthermore, the information contained in our “elementary” sub-domains like the numerical domain can be expressed as a conjunction of simple terms.

We can therefore extract the information contained in these abstract domains and model a constraint system using it. Our main requirement is to find and enumerate instances fulfilling the constraints (if any). Due to the simple nature of these constraints, we decided to employ the `Choco` constraint solver, a java library for constraint satisfaction problems (CSP) and constraint programming [JRL+08]. Another powerful alternative would have been an SMT solver.

The following Choco variables and constraints are created for the abstract state components:

5.1.1 Numerical Domain

We introduce boolean solver variables for all boolean identifiers and Integer-valued solver variables for all other numerical identifiers in the Apron abstract state. Note that integer variables impose a simplification but prevent us from finding floating point values which are contained in the *Number* type of TouchDevelop. We refrained from integrating the Ibex C++ library that would enable Choco to deal with real-values variables.

Furthermore, we add *scalar product constraints* on the identifiers for all the linear constraints in an Apron state. Because all numerical domains are convex, we get a conjunction of terms of the following form where c_i and s are constants, and v_i variable symbols:

$$\sum_i c_i \, v_i \; \otimes \; s, \qquad \otimes \in \{>, \; \geq, \; =, \; \neq\}$$

Such scalar product constraints are directly supported by Choco.

5.1.2 Invalid Domain

The validity of an identifier is a binary choice constrained by its corresponding entry in the *InvalidDomain*. The constraint generation process therefore adds a boolean solver variable for every identifier. In case the analysis determined an identifier to be *definitely valid* or *definitely invalid*, its value is fixed, otherwise decided by the solver.

5.1.3 Heap Domain

For all "points-to" choices in the heap environment, *enumerated variables* are added that may only assume a discrete set of values. For example, when a variable identifier v may point to three different heap identifiers, i.e. $env(v) = \{hid_1, hid_2, hid_3\}$, we let the solver choose any element of this set. This mainly gives us a convenient enumeration of possible variable assignments.

5.1.4 Solver Search Strategy

Choco offers a wide range of search heuristics to tune the solving process. We did not implement a custom search, but point out that the enumeration of solutions could be guided to first try different combinations for variables with a small range and few constraints, like boolean flags. This could ensure a better test coverage.

5.2 Testing

At the time of writing there was no way to automatically upload new scripts into the cloud and execute them, so they have to be entered by hand either in on the phone or the IDE in the web browser.

Since our approach requires concrete testing, we implemented a custom TouchDevelop interpreter. As the size and complexity of a comprehensive interpreter is outside the scope of this project, we provide a proof-of-concept implementation that supports basic language constructs and extended it with API methods that were needed for interesting test cases.

We outline a few design choices we made:

- **Interpretation of Sample CFG.** The interpreter operates directly on the Sample CFGs that the TouchDevelop frontend produces. It contains enough information for a concrete execution and provides a relatively high-level view of the scripts compared to the AST returned from the parser.

- **Concrete values.** We cannot reuse the infrastructure of the abstract execution, since all values there are symbolic expressions and approximations of the abstract domains. The concrete value types for primitives are implemented as thin wrappers around the corresponding Scala types, such as booleans and strings. We implement TouchDevelop's `Number` type with `Double`. Additionally, we model values of references explicitly and have a separate type for *invalid* values. Objects are simply maps of field names to values, and collections also maintain maps but with different key types such as indices for list positions.
- **Concrete heap.** The concrete heap store as well as all environments are allocated in maps. We decided to keep this state mutable to avoid excessive copying.
- **Termination.** To ensure termination, the execution has an upper limit for the number of statements which are processed. If this limit is exceeded, we report a possible non-termination.

6 Evaluation

In this section, we evaluate the new backward analysis. Our discussion includes both synthetic test cases and some published real-world scripts from the TouchDevelop cloud. For scripts derived from cloud versions or taken literally from the cloud, we indicate the publication ids. All analyses were instantiated with the polyhedra numerical domain and the option for inter-procedural backward analysis disabled, unless otherwise noted. A bug in Sample causes some persistent global variables to unnecessarily become summaries when the effect of multiple script runs is computed. This incurs a precision loss and we thus analyzed the scripts in single-execution mode that only considers one execution of a script without taking into account the persisted state between executions.

The full source code used for each example is contained in our test suite. To make automated testing possible, that source also includes annotations describing where alarms are expected and whether a definite counterexample or a false alarm should be inferred for them.

It would have been interesting to analyze a larger number of cloud scripts automatically, but this was unfortunately prevented by our concrete testing approach with a limited TouchDevelop interpreter. We were required to manually inspect the alarms found in scripts and extend the interpreter on a case-by-case basis.

6.1 Randomly Selected Cloud Scripts

We took an existing TouchDevelop test set of published cloud scripts which is already annotated with explanations of the produced alarms, including manual judgement whether they are true or false. It was previously created by Lucas Brutschy to evaluate the forward analysis. The tests were obtained by random selection of 50 scripts with the id prefix `aa`. The standard analysis of TouchBoost checks code for invalid references being used as method call targets or parameters. No other warnings were enabled and we instantiated the octagon numerical domain.

Among these 50 scripts, our analysis was not able to detect any of the 5 false alarms raised in 3 scripts. The false alarms are all except one caused by imprecision of collections in the abstract heap domain. As our collection backward semantics are even coarser, it is not possible to infer a $\perp$ entry.

However, for all the 19 scripts containing true alarms, we managed to produce at least one counterexample. Only minor changes to our analysis were required to support these scripts. The causes of the errors are mostly surprisingly simple but are very common in TouchDevelop scripts. We are thus positive that our results for this random selection represent the average case among all scripts.

We now discuss the analysis results for a few of the alarms.

6.1.1 Script `aanja`

In Listing 11, we show a script that accesses the acceleration sensor of the executing device which returns a three-dimensional vector. Such a sensor is not present on all execution platforms, for example web browsers. In this case, a call to `senses→acceleration quick` can return `invalid`.

The correct way to use this API would be to check `senses→has accelerometer` first and inform the user that the script does not support his device if there is no acceleration sensor. However, the script directly uses the result by scaling it to set parameters of the game engine.

Our analysis state keeps a boolean flag in the `senses` singleton object that indicates the presence of the acceleration sensor. Initially, the value of this flag is undetermined, causing the forward analysis to consider the possibility of an invalid value returned from sensor readings. In the line `a := acc->scale(200)`, the implicit assertion `acc != invalid` fails to hold in general and causes an alarm.

When we investigate this abstract error with the backward analysis, we assume the negation of this assertion, i.e. we start with $acc ==$ `invalid`. This means `senses→acceleration quick` did indeed return `invalid`. After propagating back this fact, we can conclude that the error occurs when starting from an initial state with the accelerometer capability flag set to $false$. A simple run in the interpreter confirms that this is indeed a definite counterexample.

The same reasoning is applied for an alarm on line 11 which has the exact same cause, but we actually never run into it because the execution already terminates in the first error.

```
 1  ...
 2  event gameloop() {
 3   acc := senses->acceleration_quick;
 4   a := acc->scale(200);
 5   data->board->set_gravity(a->x, a->y);
 6   data->board->evolve;
 7   data->board->update_on_wall;
 8   // Add game logic here
 9   // Get the acceleration and rescale to pixels.
10   accp := senses->acceleration_quick;
11   p := accp->scale(800);
12   // Assign acceleration as board gravity.
13   data->board->set_gravity(p->x, p->y);
14   // Apply physics
15   data->board->evolve;
16   // Redraw board on wall
17   data->board->update_on_wall;
18  }
19  ...
```

Listing 11: Script `aanja`: Accessing missing acceleration sensor

6.1.2 Script aaoweirj

Listing 12 shows a TouchDevelop toy example of which there exist many variations. The script randomly chooses a song from the media collection on the phone and plays it in reaction to the user changing the orientation of his phone. The problem with this code is that it does not handle the case where the media collection does not contain any sound files. If this is true, the `random` call on the collection returns an `invalid` value, but the invocation of `play` on this result immediately causes the script to crash.

The forward analysis fails to prove that the call target of `play` must be valid and therefore correctly reports an alarm. Our backward analysis now works backwards from this call with the assumption that the return value of the random is `invalid`. It determines that this may only happen if the song collection size is 0. As the script does not modify the collection between the start of the event and this program point, the backward analysis results in a refined entry state with this information. We then generate a concrete counterexample with an empty collection, and determine that it is indeed a true alarm.

Note that this is an example where our lack of precision for collection operations and contents has no negative effect because all that is relevant for the error is the collection size.

```
action main() {
  "TouchDevelop is cool!"->post_to_wall;
}

event phone_face_down() {
  // possible crash here!
  s := media->songs->random->play;
  phone->vibrate(0.6);
  wall->set_background(colors->random);
}
```

Listing 12: Script `aaoweirj`: Accessing an empty song collection

6.1.3 Script aaib

The script named “Is it up?” displayed in Listing 13 asks the user for a web address and consults a web service to check whether the given site is up for other people.

TouchBoost produces several (non-spurious) alerts for this script:

- `web->download(url)` issues a HTTP request and returns the result in a string internet. This operation may fail depending on the connectivity and the reachability of hosts. In such a case, the method returns `invalid`. In line 15, the string is passed to `web->json` that parses JSON. The analysis detects that this argument may be `invalid`.
- `web->json` may either return a valid JSON collection object or `invalid` if the input string is not proper JSON. However, the result is directly used without a check: `json->field(”status_code”)`
- Since we do not have any specifications or guarantees about the accessed web service, the JSON object may contain anything. Even if variable `json` is valid, all unchecked member value usages like `json->field(”status_code”)->to_number` immediately cause the script to crash if the fields are not present or have the wrong type. TouchBoost warns in all 4 instances of this .

Our backward analysis is able to handle the first error: The analysis adds a nondeterministic string variable for the result of the web download. We infer its contents to be `invalid` and a counterexample is generated with this input. The concrete interpreter then immediately runs into the first error.

On the other hand, we are not able to construct a counterexample in the case where both the web download and the JSON parsing succeed but the field accesses fail. We would have to come up with a valid JSON collection object that can contain anything but a given field as our abstract domains are not expressive enough to describe what a collection does definitely *not* contain - in constrast to what it *may* or *must* contain.

```
1  ...
2  action isitup() {
3   // Utilize isitup.org json API to check sites are working.
4   // A first attempt at handling json data.
5   //
6   // Site to check.
7   site := wall->ask_string(”Site to check.”);
8   //
9   // Build the query url.
10  url := ”http://isitup.org/” || web->url_encode(site) || ”.json”;
11  //
12  // Download the result and parse it into a json data structure
13  downloaded := web->download(url);
14  json := web->json(downloaded);
15  // Fetch required fields from the json object.
16  sc := json->field(”status_code”)->to_number;
17  dom := json->field(”domain”)->to_string;
18  if sc = 1 then {
19    // If site is up then display some info.
20    ip := json->field(”response_ip”)->to_string;
21    rc := json->field(”response_code”)->to_string;
```

```
    wall->prompt(dom || " (IP: " || ip || ")" || "\nIs working.");
  }
  else {
    // If site is down then display.
    wall->prompt(dom || " Is not working.");
  }
...
```

Listing 13: Script `aaib`: Using HTTP APIs without checks

6.2 Counterexamples for Numerical Errors

We demonstrated above that our analysis supports reasoning about invalid values, but interesting errors also include the violation of numerical bounds. We now discuss one such test case in detail.

6.2.1 Motivating Example

We started out with the code in Listing 14 as a motivating example for our work. It is a variation of the real-world script action shown in Listing 15 (id `hwyo`). The alarms detected in the script are all triggered by particular interactive user inputs. As such, it is a good example of our handling of non-determinism. Both alarms are non-critical in the sense that TouchDevelop does not crash, but the behavior may be undesirable and unexpected to the user.

For the error where there picture size parameters in the call

`media->create picture(w, h)`

are not validated, the analysis simply infers a definite counterexample with negative user inputs. The values of the relevant identifiers as determined by the solver are:

Id	Value	Description / Origin
$nondet_1$	-50000	picture width, `ask_number(‘‘Width?’’)`
$nondet_2$	0	picture height, `ask_number(‘‘Height?’’)`

where $nondet_i$ denote the non-deterministic identifiers that we introduce in the state to describe user inputs. The second error is less straightforward as it involves conditional non-deterministic access in a loop. The statement

`pic->draw text(math->rand(w), math->rand(w), text, font, 0, colors->rand)`

is only executed when the user choose the second drawing style. Text may be drawn out-of-bound because the range of the y coordinate is wrong, as the code should read `math->rand(h)`. But for the error to happen the random number generator needs to yield a high enough number, i.e. `math->rand(w)` $> h$. The counterexample found by our analysis is:

Id	Value	Description / Origin
$nondet_1$	50001	picture width, `ask_number(‘‘Width?’’)`
$nondet_2$	12500	picture height, `ask_number(‘‘Height?’’)`
$nondet_3$	false	boolean flag, `ask_number(‘‘radial shape?’’)`
$nondet_4$	25000	random x coordinate, `math->random(w)` (unrolled iteration)
$nondet_5$	25001	random y coordinate, `math->random(w)` (unrolled iteration)
$nondet_6$	0	random x coordinate, `math->random(w)`(loop summary)
$nondet_7$	0	random y coordinate, `math->random(w)` (loop summary)

It leads to the undesired behavior in the first iteration of the loop. Note that $nondet_4$ and $nondet_5$ are identifiers generated for the random numbers obtained in the first loop iteration, while $nondet_6$ and $nondet_7$ are separate summary identifiers used to capture the non-deterministic decisions in the remaining iterations. The loop unrolling is critical here as we are only able to infer a suitable value for $nondet_5$ because it is determined to be non-summary.

```
action draw(text: String, font: String) {
  var w := wall->ask number(”width?”)
  var h := wall->ask number(”height?”)
  var radial := wall->ask boolean (”radial shape?”)
  var pic := media->create picture(w, h)
  for 0 <= i1 < 50 do {
    if radial then {
      pic->draw text(w/2, h/2,
            text, font, math->rand(360), colors->rand)
    } else {
      pic->draw text(math->rand(w), math->rand(w),
            text, font, 0, colors->rand)
    }
  }
}
```

Listing 14: Motivating example

```
// Textmaster, Circler, Waller and Starouz: Copyright Pouya Animation Inc
// License: GNU GPL 3
action Waller() {
  // this code will get a text from user
  wall->create_text_box(”note: Waller will make a wallpaper from a text,
    that this text will write in a some random places,
    so now you should enter that text.”, 18)->post_to_wall;
  $text := wall->ask_string(”Enter a text to make your wallpaper”);
  wall->clear;
  // this code will get a font size from user
  wall->create_text_box(”note: Waller will make a wallpaper with some
     texts that the texts have a muximum font size and now you should
    write this maximum size", 18)->post_to_wall;
  $font := wall->ask_number("Please write a maximum size for font");
  // This code will ask user a number for next loop
  // (that how many user need text)
  wall->clear;
  wall->create_text_box("note: Waller will put some texts in some places
        now hamany text you want?", 18)->post_to_wall;
  $i := wall->ask_number("how many text you want?");
  // This code will ask users screen size
  wall->clear;
  $w := wall->ask_number("enter your phones screens width");
  // Will ask users screens height
  wall->clear;
  $H := wall->ask_number("Enter your screens height");
  // This is code will create a picture for Walling
  wall->clear;
  wall;
  $pic := media->create_picture($w, $H);
  $pic->post_to_wall;
  // This code will drow texts in their places.
  for 0 <= i1 < $i do {
   $pic->draw_text(math->rand($w), math->rand($w),
    $text, $font, math->rand(360), colors->rand);
   $pic->update_on_wall;
  }
  // This Code will ask user about saving picture
  wall->clear;
  if wall->ask_boolean("waller@termini :~$", "do you want to save it?
    it will save in your photo gallery") then {
   $pic->save_to_library;
  }
}
```

Listing 15: Extract from script hwyo

6.3 Detection of False Alarms

As noted earlier, if we infer an entry state that is $\perp$, we may safely conclude that the analyzed alarm is false. This happens when the backward analysis determines that all paths to the error location are not viable.

We now describe a few cases where we managed to detect such false alarms.

6.3.1 Imprecision Due to Joins

Missing disjunctive information is common theme when analyzing programs that perform case distinctions to produce results. The action in Listing 16 is adapted from [Riv05] where it serves as a motivating example. It simply computes the absolute value of a number and performs an assertion, yet the forward analysis fails to verify it – even with the most precise, available polyhedra domain. The assertion amounts to showing $abs(x) > 5 \implies x < -5 \lor x > 5$ which clearly holds.

The forward analysis infers that we must have $x > 0 \land y = x$ at the end of the true branch and $x \leq 0 \land y = -x$ after the else branch, respectively. However, after the `if` statement that assigns y, a join is performed and results in $y \geq x$. This fact is useful but not strong enough to establish the assertion. Ideally, our numerical domains would be able to represent the disjunction $(x > 0 \land y = x) \lor (x \leq 0 \land y = -x)$ when doing the join, but this lies beyond their expressiveness because they are all convex.

The backward analysis starts in the error state $y > 5 \land -5 \leq x \leq 5$ at the assertion statement and propagates this state back into the if-branches. In both branches, the greatest lower bound is taken with the states from the forward analysis and the numerical domain immediately determines that the states there must be $\bot$, i.e. definitely not forward- and backward reachable at the same time.

```
action abs(x: Number) returns (y: Number) {
    if (x > 0) then {
      y := x;
    } else {
      y := -x;
    }

    if (y > 5) then {
      //:: ExpectedOutput(assert.failed)
      contract->assert(x < -5 or 5 < x, "must hold");
    }
}
```

Listing 16: Action computing absolute value: False alarm because of imprecise join (`join_imprecision.td`)

6.3.2 Imprecision Due to Widening

Another notorious cause of imprecision is the widening operation. It accelerates and ensures the termination of the fixed point iteration for abstract domains that are lattices of infinite (or prohibitively large) height. In Sample, we apply widening at the entry of CFG basic blocks whenever the iteration encounters them more often than a fixed number of times (see Algorithm 1).

The problem with widenings is that they “over-jump” the least fixed point in the abstract domains lattices. Listing 17 display a loop where this happens. We use the default setting of Sample that widens the state after 3 iterations of the loop. Instead of $x = 11$, we only know that $x > 10$ after the loop. Assertions that depend on the number of iterations performed cannot be shown to hold, for example $x \leq 11$ which directly refers to the loop counter.

The backward analysis of this alarm tries to find executions that lead to $x > 11$, but as it enters the loop body backwards, it determines that $x \leq 11 \wedge x > 11 = \perp$. This happens in all backward iterations, and together with $x = 0$ at the start the entry state also becomes $\perp$.

```
action main() {
    var x := 0;
    while (x <= 10) do {
     // .. other logic ..

     x := x + 1;
    }

    // assert fails due to widening...
    contract->assert(x <= 11, ”must hold”);
}
```

Listing 17: Loop with widening imprecision (`wideningAlarm.td`)

6.3.3 Interprocedural Detection of False Alarm

Some false alarms occur in an interprocedural setting. We identified a common pattern in published TouchDevelop scripts that causes problems in our forward analysis: At some point during the execution, a bunch of global state is initialized. This state is accessed in the events of the script, but the programmer has only limited control over when they may be triggered. To prevent access to uninitialized, invalid state, all accesses are wrapped with a check of a boolean initialization flag.

One popular script with this pattern is the game `CloudHopper` (script id `wbxsa`). TouchBoost produces several false alarms regarding property accesses on invalid targets. Some of them are caused by collection imprecisions which we cannot handle. Furthermore, the script is quite complex and uses API calls we did not implement in the interpreter. We therefore extracted the relevant parts related to the alarm with the initialization of global data, shown in Listing 18.

The analysis is unable to show that the access in line 15 is always safe because it cannot relate the flag $data{\to}init$ to the validity of $data{\to}board$. Once more, we suffer from missing disjunctive domains; Our states cannot express $data{\to}init \implies data{\to}board \neq invalid$.

The interprocedural backward analysis then tries to find the initial states before the execution of `main` that lead to this situation with $data \rightarrow init \wedge data \rightarrow board = invalid$. Theoretically, there is an unbounded number of possible event sequences to examine but they must all follow the informal execution pattern

$$main \Rightarrow (gameloop \mid shake)^* \Rightarrow gameloop$$

as the execution starts in `main` and ends at the error in `gameloop`, with an arbitrary number of events triggered in between. We compute the backward semantics by taking the fixed point of these repeated events. The analysis detects that all code paths from the start to the error must have set the initialization flag, but in order for that to happen, the initialization code would also have been executed. The state thus becomes $\perp$.

```
var init: Boolean
var sprites: Sprite_Set
var board: Board
// .. more resources

action main() {
 // all global data uninitialized, defaulting to invalid/false
}

event gameloop() {
   if (data->init) then {
      // game logic ..

      // failing assertion: data->board might be invalid?
      data->board->evolve();
   }
}

event shake() {
   if (not data->init) then {
      data->board := media->create_full_board();
      data->init := true;
   }
}
```

Listing 18: Interprocedural false alarm

7 Conclusion

In this thesis, we extended the static analyzer TouchBoost with backward analysis functionality in order to investigate the alarms reported for TouchDevelop scripts. The new analysis permits the computation of refined entry states (program inputs) which narrow down the program executions that may lead to a reported error. As a particular application of this backward analysis, we concretized these resulting abstract entry states and performed concrete testing with an interpreter to infer *definite counterexamples*. We showed that in some cases, the resulting states also allow us to identify false alarms caused by over-approximation of the program semantics.

Both the theoretical foundations of the approach based on abstract interpretation and the changes needed in the TouchBoost abstract semantics were presented. The implementation aims to be generic in the sense that new abstract domains could easily be extended with the necessary backward semantics. We note that the backward analysis may also be useful in other contexts than automated counterexample generation, namely whenever an automated, sound backward propagation of information towards the execution entry of a program is needed.

Furthermore, we demonstrated that our analysis produces promising results for the error exploration in a range of small but real-world TouchDevelop scripts. We are thus positive that our work helps potential users get a better understanding of the root causes of TouchBoost alarms in their scripts.

7.1 Related Work

Backward abstract interpretation and also refining analyses date back to the original works of Cousot and his PhD thesis [Cou78]. The presentation in [Riv05] served as the basis of our work. The authors explore techniques to better understand the origin of alarms in the Astree analyzer [CCF+05] and to help with the manual classification of alarms as true or false. Their work includes trace partitioning for loops and a syntactic program slicing technique which we both did not implement. On the other hand, our goal was to go a step further after the backward analysis and produce definite counterexamples when possible.

In more recent work, Cousot et al. [CCL11, CCFL13] introduce a symbolic backward analysis for contract inference that yields necessary preconditions. In contrast to the method used here, they over-approximate the *good* runs which do not lead to errors. The complement of the resulting entry state is thus *sufficient* for errors to occur. It would be interesting to combine this approach with the traditional technique we use and intersect that complement with the refined entry states that we obtain.

Brauer [BS12] take another approach and try to directly construct counterexample traces by finding paths to the program entry with the help of SAT-based algorithms. The technique represents the program state with boolean formulas (e.g. bit vectors for integer variables) and is based on abduction of propositional boolean logic. In the presence of loops, multiple loop iterations are summarized with a boolean transformer, which may lead to an under-approximation of the transition relation. Like the refining backward analysis, Brauer also reuses the results from the forward analysis.

[Min12] presents an under-approximating abstract interpretation with the polyhedra numerical domain to infer sufficient preconditions that make sure the program always stays within a set of safe states. They show that the approach can also be adapted to infer sufficient conditions for the program to either not terminate or fail in an error state. Its very strong focus on the polyhedra abstract domain kept us from adopting the approach in this thesis.

7.2 Open issues and Future Work

At last, we summarize a few weaknesses in the current implementation and improvements that could be made to fix them, as part of future work.

- **Improve testing of potential counterexamples.** We had to implement our own TouchDevelop interpreter to exercise tests of scripts with given inputs since we had no means to upload and run scripts automatically on the Microsoft cloud infrastructure. Due to the complexity of the language and API, our interpreter remains a proof of concept and implements only a tiny subset of all features. If the TouchDevelop team decides to open up their web APIs to allow external code upload, we could replace the testing component. A further consideration is the use of more elaborate techniques like concolic testing instead of randomly picking candidates.
- **Graphical error exploration.** Currently there is no user interface to explore errors; all alarms are investigated and messages on the console indicate the outcomes. It would be nice if a user could select an alarm and start the backward analysis for it. Input states that are counterexamples could then be rendered graphically.
- **Better collection support.** Many errors are caused by wrong assumptions about collections and their contents. Precise backward reasoning for these types of errors should be implemented which would require the introduction of new abstract domains that are able to express that certain elements must not be contained in the collections.

7.3 Acknowledgements

I would like to thank my supervisor Lucas Brutschy for his helpful advice throughout the thesis, and Prof. Dr. Peter Müller for giving me the opportunity to work on this challenging project in his group.

Furthermore, I am indebted to Severin Heiniger who was also working on a Sample project and sharing the office with me. We had many fruitful discussions about the Sample architecture. His countless refactorings enabled us both to create automated end-to-end test suites for our projects which greatly helped with the development.

A Notation Overview

Concrete program states :	Σ
Concrete Domain (Lattice) :	$(D = \mathcal{P}(\Sigma),\ \sqsubseteq,\ \sqcup,\ \sqcap,\ \bot,\ \top)$
Abstract Domain (Lattice) :	$(D^\sharp,\ \sqsubseteq^\sharp,\ \sqcup^\sharp,\ \sqcap^\sharp,\ \bot^\sharp,\ \top^\sharp)$
Abstraction :	$\alpha : D \to D^\sharp$
Concretization :	$\gamma : D^\sharp \to D$
Galois connection :	$D \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} D^\sharp$
Initial program states :	$\mathcal{I}$
Final (erroneous) program states :	$\mathcal{E}$
Concrete program semantics :	$\hat{\mathcal{C}}_{\mathcal{I}} : ProgramPoint \to D$
Abstract forward program semantics :	$\hat{\mathcal{F}}_{\mathcal{I}} : ProgramPoint \to D^\sharp$
Abstract backward program semantics :	$\hat{\mathcal{B}}_{\mathcal{E}} : ProgramPoint \to D^\sharp$
Abstract forward-backward refined program semantics :	$\hat{\mathcal{B}}_{\mathcal{E}}^{ref} : ProgramPoint \to D^\sharp$
Concrete transfer functions :	$\overrightarrow{c}[\![s]\!] : D \to D$
Abstract forward transfer functions :	$\overrightarrow{f}[\![s]\!] : D^\sharp \to D^\sharp$
Abstract backward transfer functions :	$\overleftarrow{b}[\![s]\!] : D^\sharp \to D^\sharp$
Abstract refining backward transfer functions :	$\overleftarrow{b^{ref}}[\![s]\!] : D^\sharp \times D^\sharp \to D^\sharp$

B Infrastructure Work

Apart from the work concerned directly with the thesis topic, a significant effort has been spent on improving the infrastructure of Sample and the TouchBoost project. The goal was to boost the productivity when working with the Sample code base.

Most of the following was done in collaboration with Severin Heiniger.

B.1 Move to Mercurial and SBT

Version control was switched from the aging Subversion to Mercurial (Hg). As several people were working on the source code at the same time, we made heavy use of its flexible branching and merging functionality.

The Sample project configuration used to be entirely IntelliJ-based. Compiling and executing programs always had to be performed using the GUI. With SBT (Scala Build Tool), we introduced a proper build tool for Sample. It is able to manage library dependencies and generate IntelliJ project files automatically. Compilation and testing can now be automated with simple commands.

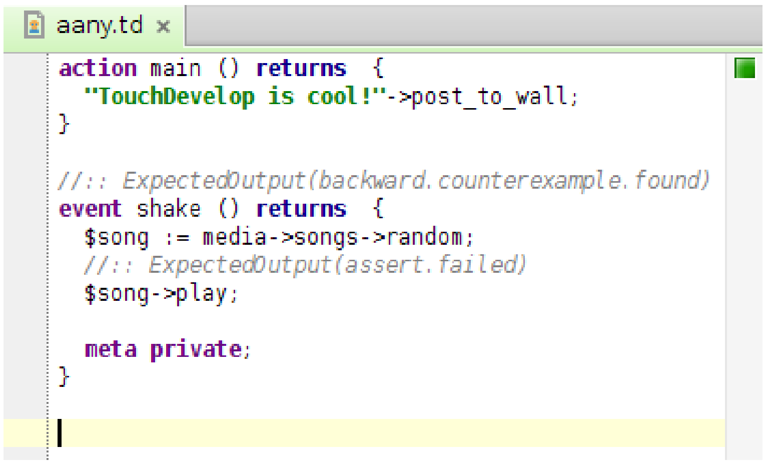
B.2 End-to-End Tests for TouchBoost

We integrated the existing annotation parser of the `SIL` project written by Stefan Heule which was adapted by Severin Heiniger to allow the specification of expected analysis results in the source code of TouchDevelop test cases.

This enables automated end-to-end testing for TouchBoost, which was not possible before and hopefully contributes to the robustness of the tool.

B.3 Syntax highlighting

To make working with TouchDevelop scripts locally more visually appealing, we defined a new syntax highlighting scheme for IntelliJ. Operations such as commenting out pieces of code are also enabled by that change. An example can be seen in Figure 8.



```
action main () returns  {
  "TouchDevelop is cool!"->post_to_wall;
}

//:: ExpectedOutput(backward.counterexample.found)
event shake () returns  {
  $song := media->songs->random;
  //:: ExpectedOutput(assert.failed)
  $song->play;

  meta private;
}
```

Figure 8: IntelliJ syntax highlighting for TouchDevelop scripts

References

[Bon13] Y. Bonjour. Must analysis of collection elements. Master's thesis, ETH Zurich, 2013.

[Bou93] François Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Formal Methods in Programming and their Applications*, pages 128–141. Springer, 1993.

[BS12] Jörg Brauer and Axel Simon. Inferring definite counterexamples through under-approximation. In *NASA Formal Methods*, pages 54–69. Springer, 2012.

[CC77] 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 the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 269–282. ACM, 1979.

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *The Journal of Logic Programming*, 13(2):103–179, 1992.

[CCF$^+$05] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The astrée analyzer. In *Programming Languages and Systems*, pages 21–30. Springer, 2005.

[CCFL13] Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. Automatic inference of necessary preconditions. In *Verification, Model Checking, and Abstract Interpretation*, pages 128–148. Springer, 2013.

[CCL11] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. Precondition inference from intermittent assertions and application to contracts on collections. In *Verification, Model Checking, and Abstract Interpretation*, pages 150–168. Springer, 2011.

[CFC11] Giulia Costantini, Pietro Ferrara, and Agostino Cortesi. Static analysis of string values. In *Formal Methods and Software Engineering*, pages 505–521. Springer, 2011.

[Cou78] Patrick Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes.* PhD thesis, Institut National Polytechnique de Grenoble-INPG, 1978.

[Cou97] Patrick Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Electronic Notes in Theoretical Computer Science*, 6:77–102, 1997.

[Cou98] Patrick Cousot. Calculational design of semantics and static analyzers by abstract interpretation. *NATO Int. Summer School*, pages 83–94, 1998.

[FFJ12] Pietro Ferrara, Raphael Fuchs, and Uri Juhasz. Tval+: Tvla and value analyses together. In *Software Engineering and Formal Methods*, pages 63–77. Springer, 2012.

[FSB14] P. Ferrara, D. Schweizer, and L. Brutschy. Touchcost: Cost analysis of touchdevelop scripts. In *Fundamental Approaches to Software Engineering (FASE)*, 2014. to appear.

[JM09] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *Computer Aided Verification*, pages 661–667. Springer, 2009.

[JRL+08] Narendra Jussien, Guillaume Rochart, Xavier Lorca, et al. Choco: an open source java constraint programming library. In *CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08)*, pages 1–10, 2008.

[Mic] Microsoft. Touch Develop environment. http://www.touchdevelop.com/. [Online; accessed 06-April-2014].

[Min06] Antoine Miné. Symbolic methods to enhance the precision of numerical abstract domains. In *Verification, Model Checking, and Abstract Interpretation*, pages 348–363. Springer, 2006.

[Min12] Antoine Miné. Inferring sufficient conditions with backward polyhedral under-approximations. *Electronic Notes in Theoretical Computer Science*, 287:89–100, 2012.

[Riv05] Xavier Rival. Understanding the origin of alarms in astrée. In *Static Analysis*, pages 303–319. Springer, 2005.

[TMdHF11] Nikolai Tillmann, Michal Moskal, Jonathan de Halleux, and Manuel Fahndrich. Touchdevelop: Programming cloud-connected mobile devices via touchscreen. In *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*, pages 49–60. ACM, 2011.

[ZFC12] Matteo Zanioli, Pietro Ferrara, and Agostino Cortesi. Sails: static analysis of information leakage with sample. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 1308–1313. ACM, 2012.