Master research Internship

Internship report

Automatic equivalence verification for translation validation

Domain: Programming Languages - Logic in Computer Science

Author:
Alexandre Drewery

Supervisor:
David Pichardie
META
Thomas Jensen
INRIA EPICURE Team

Abstract: In the context of codebase modernization, there is a need for an automated process of validation of code translation. This can be achieved by checking program equivalence between the old and the new code. We study this problem in the context of Java Bytecode programs for Java to Kotlin translation validation.
In a bibliographic preliminary study we review various techniques for program equivalence. During the internship we focus on equality saturation, a solver-like approach. We confront Peggy, an existing equivalence checker to the specific context of Java to Kotlin conversion, and improve its expressivity. We also propose an improved algorithmic framework for the technique to support a new kind of equivalence rules.

Contents

Introduction

Over the span of the life of a codebase in the industry, new programming languages appear and improve on older ones with new features and more safety. For instance, Kotlin is a recent programming language designed as a modernization of Java. It implements an expressive type system that allows for greater concision and safety of programs. Some tech companies chose it as their preferred language of development over Java and decided to modernize their codebase to Kotlin.

In the context of industry, these codebases are around millions of lines of code long. However, since Kotlin was designed to be interoperable with Java (both languages are compiled into Java Bytecode), a gradual translation is possible. An automatic translation tool exists but is insufficient: developers have to translate almost by hand. Not only does this process not scale to the size of the codebase, but it also is error-prone. This is why some form of translation validation is needed.

The simplest way to perform this validation is to check if the original Java program and its Kotlin translation are compiled into identical bytecodes. However, since the bytecodes are produced by two different compilers, they are likely to feature many unimportant differences such as ordering of independent blocks of instructions, naming of local variables, etc. The translation validation we seek is semantic in nature: ensuring that the two bytecodes have equivalent behaviours (the bidirectionnal arrow on Figure 1).

Unfortunately, program equivalence is undecidable in a general context [13]: it might seem that a semantic translation validation is impossible. The classical approach to solve this problem is to propose a sound but not complete verifier: the equivalence checker will never report a semantic equivalence if the two programs are not equivalent, however it is not supposed to be able to detect all pairs of equivalent programs. In the context of industry codebases modernization, this solution makes sense: a translation validated by the equivalence checker can be trusted; and if the equivalence checker is able to validate enough translations, the amount of code that has to be reviewed by human is significantly reduced.

The goal of this internship is to develop a program equivalence technique for bytecodes of a Java class and a candidate Kotlin translation.

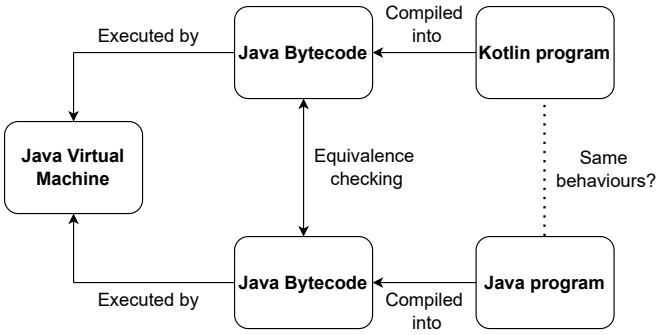


Figure 1: Overview of the equivalence checking problem for Java-Kotlin conversion.

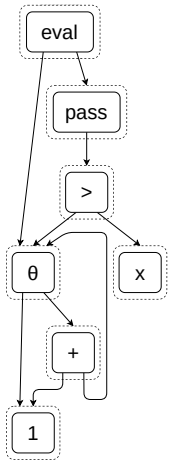
In winter, before the beginning of the internship, we reviewed existing program equivalence techniques and compared their strengths and weaknesses. We overall reviewed two families of approaches: equivalence checkers based on program logic and solver-like techniques based on an algorithm called equality saturation.

Our first contribution is a set of experiments studying various Java and Kotlin programs and their associated bytecodes. We were interested in what kind of differences were introduced by the automated IntelliJ Java-to-Kotlin conversion and by the use of two different compilers. We also tried Peggy [17], a program equivalence tool based on the equality saturation approach: we confronted its expressivity to our specific problem. We discovered a variety of differences between Java and Kotlin bytecodes, some of which Peggy could not prove equivalent right away: for instance, Figure 2a represents the pseudo-codes of the two bytecodes obtained by compiling two very similar loop-for programs in Java and Kotlin.

```
1  #in Java
2  i = i0
3  while (i <= ifinal) {
4      loop body
5      increment i
6  }
```

```
1  #in Kotlin
2  i = i0
3  if (i <= ifinal) {
4      loop body
5      while (i != ifinal){
6          increment i
7          loop body
8      }
9  }
```

(a) Pseudo-codes for a Java-Kotlin compilation difference for loop-for.



(b) Peggy representation of the Java program.

Figure 2: Some of the concepts studied in this internship.

Peggy is very generic and can be adapted to a problem without having to modify its internal algorithmic core. Contrarily to other checkers of the bibliography, Peggy was also available online. In addition, we found significant differences between control flows in Java programs and Kotlin programs. During the bibliographic study, the equality saturation approach was shown to be effective in those cases. This technique in general can also be placed in the active study field of e-graph manipulation [18]. For all those reasons, we decided to focus the remainder of the internship on equality saturation and the Peggy tool. This required us to become more familiar with the PEG and EPEG program representations of Peggy: an example EPEG for the Java program of Figure 2a is shown in Figure 2b.

Our second contribution was to improve Peggy's expressivity by allowing it to prove equivalence for a family of programs that presented a challenging loop-for structure. We studied in depth the terms associated to these programs and designed a set of equivalence rules that would allow to deal with them.

Peggy's performance are highly dependant on the proof length, and the proofs for these challenging programs can be quite long. Our last and main contribution is an algorithmic framework that aims to improve equality saturation by allowing it to use a new kind of equivalence rules: contextual equivalence rules. These more powerful rules would reduce the number of rules needed to perform a proof. While standard equivalence rules can only work on a local part of the control flow of programs, our contextual equivalence rules are able to use contextual knowledge from all the control flow.

The report of the bibliographic study can be found in Part I. In Section 6, we present our experimental survey of Java-Kotlin differences and the tools we used. Section 7 gives the background concepts needed to work with Peggy EPEGs. In Section 8 we focus on the new equivalence rules we designed to allow Peggy to perform in the Java-Kotlin context. Lastly, our new algorithmic framework for contextual rewriting is presented in Section 9 and we discuss it and its possible extensions in Section 10.

To the attention of our reviewers: the internship report of Part II is self contained: a detailed rereading of Part I is overall non necessary. Whenever a concept of the bibliography study is needed in the report, we either give a rough summary of the concept or give a more in depth explanation than we did in the bibliography (notable in Section 7). In Part I, new content is indicated in colour, but we also modified some parts to take into account reviews.

Part I
Bibliographic study

In this bibliographic report we review some state of the art articles about the problem at hand. Section 1 focuses on Java bytecode. Section 2 presents notions of relational program logic, notably relational Hoare logic. These concepts are used by some of the program logic verifiers presented in Section 3. In Section 4, we review an equational decision procedure based on the equality saturation method.

1 Java bytecode

The Java bytecode language or Java virtual machine language (JVML) is the instruction set of the Java virtual machine (JVM). This bytecode is the representation of compiled Java and Kotlin programs. In the context of this bibliographic report we introduce a small subset of bytecode language to serve as an example for the studied concepts. We use a simplified syntax inspired by Freund and Mitchell bytecode typing works [10].

Among its characteristics, JVML is stack-based. We give an example of small bytecodes associated to two programs in Figure 3. Typical stack manipulations include operands of calculations being pushed on the stack, data being popped from the stack when used by another instruction, or the top of the stack being stored in memory.

JVML also contains typing information that allows the virtual machine to run some consistency checks and detect type errors prior to execution. For pedagogical reasons, we restrict our small bytecode language to integer manipulation so any type checking is trivial.

```
(y = 0) ? (x := z) : (x := 5)
```

```
1 push 0
2 load y
3 ifeq 7
4 push 5
5 store x
6 return
7 load z
8 store x
9 return
```

(a) Program 1.

```
(y = 0) ? (x := 4) : (x := z)
```

```
1 push 0
2 load y
3 ifeq 7
4 load z
5 store x
6 return
7 push 4
8 store x
9 return
```

(b) Program 2.

Figure 3: Two pseudocode Java programs and associated bytecodes.

We present an operational semantics for bytecode in order to work formally on it. For the time being we focus on the code of a single method without calls or exception throwing and a very limited instruction set manipulating local variables.

We describe the state of the virtual machine during execution as a triple $\langle pc, \sigma, \rho \rangle$ where:

- pc is the program counter position, pointing to the next instruction to be executed;
- σ is the operand stack. $\sigma[i]$ refers to the ith element of the stack, starting from the top which is $\sigma[0]$. Notation $v :: \sigma$ denotes stack σ on top of which the value v is pushed;

- ρ is a function mapping local variables to their values. We denote $\rho[x \to v]$ for the function in which x now maps to value v and other variables y still map to their previous value $\rho(y)$.

We use the definition of operational semantics for bytecode of Freund [10] in Figure 4 to define a small-step judgement $\to$.

Instruction at pc	Condition	State	Next state
push v		$\langle pc, \sigma, \rho\rangle$	$\langle pc+1, v :: \sigma, \rho\rangle$
pop		$\langle pc, v :: \sigma, \rho\rangle$	$\langle pc+1, \sigma, \rho\rangle$
add		$\langle pc, v_1 :: v_2 :: \sigma, \rho\rangle$	$\langle pc+1, v_1 + v_2 :: \sigma, \rho\rangle$
load x		$\langle pc, \sigma, \rho\rangle$	$\langle pc+1, \rho(x) :: \sigma, \rho\rangle$
store x		$\langle pc, v :: \sigma, \rho\rangle$	$\langle pc+1, \sigma, \rho[x \to v]\rangle$
ifeq L	$v_1 = v_2$	$\langle pc, v_1 :: v_2 :: \sigma, \rho\rangle$	$\langle L, \sigma, \rho\rangle$
ifeq L	$v_1 \neq v_2$	$\langle pc, v_1 :: v_2 :: \sigma, \rho\rangle$	$\langle pc+1, \sigma, \rho\rangle$
goto L		$\langle pc, \sigma, \rho\rangle$	$\langle L, \sigma, \rho\rangle$

Figure 4: Operational semantics for a fragment of bytecode from [10]. We define the small-step judgement $s \to s'$ if state s' follows state s due to one of the rules of this operational semantics.

We define a big-step judgement $\Rightarrow$ from this small-step semantics as follows: we say that a state s is *final* if its program counter has reached a *return* instruction. We denote $s \Rightarrow s'$ (a full run of the program takes state s to state s') if and only if $s \to^* s'$ and s' is final.

As an example for program 1 of Figure 3, starting from a state in which stack is empty and the defined locals are $y = 0$ and $z = 4$, we have the following small steps:

$$\langle 1, [], \rho\rangle \to \langle 2, [0], \rho\rangle \to \langle 3, [0,0], \rho\rangle \to \langle 7, [], \rho\rangle \to \langle 8, [4], \rho\rangle \to \langle 9, [], \rho[x \to 4]\rangle$$

and the bigstep judgement $\langle 1, [], \rho\rangle \Rightarrow \langle 9, [], \rho[x \to 4]\rangle$.

Our bytecode fragment is deterministic, meaning that for all states s there is at most one state s' such that $s \to s'$ (and therefore at most one s_{final} such that $s \Rightarrow s_{final}$).

Note that some situations can lead to a program being stuck without having reached the end of its code: for instance trying to *pop* when the stack is empty. Additionally the fragment of JVML we described is very small. This fragment can later be extended by adding richer types, method calls, object-oriented features, exception throwing, etc. Freund and Mitchell propose a semantics for a large fragment of bytecode as well as a type system that is sound: any well-typed program does not get stuck during its execution [10]. This work paves the way for further verification results about Java Bytecode: during the internship, we aim to provide a semantic soundness proof of our program equivalence analysis.

2 Relational Hoare Logic

Program logics are a collection of formal methods to specify and prove properties about programs. For instance, Hoare logic [12] allows to express statements such as $\{P\}\ C\ \{Q\}$: from a state satisfying precondition P, program C either does not terminate, or terminates and yields a state that satisfies postcondition Q.

Relational Hoare logic (RHL) [6, 7] is an axiomatic approach to program comparison. It is a variant of Hoare logic that aims to reason about two programs at once. As it studies two programs rather than one, it needs to track and compare the states of two executions: we use **binary relations** instead of properties for the precondition P and the postcondition Q of the program. In order to get a better understanding of how this technique can be of use for our problem, we designed a basic core of RHL for our fragment of Java Bytecode from the Section 1.

2.1 Relational assertions

As it is the case with properties in standard Hoare logic, we need to specify a syntax for our binary relations. We do so as follows:

- **arithmetic expressions** over integers $n \in \mathbb{N}$, variables x from a set X and values on the stacks Σ (indices 1 or 2 indicates which program the variable or stack value is from, and we are able to pinpoint the ith value in a stack):

$$E := n \mid x_1 \mid x_2 \mid \Sigma_1[i] \mid \Sigma_2[i] \mid E \ + \ E$$

  An example of arithmetic expression: $y_1 + \Sigma_1[0]$;

- **relational assertions** are defined by conjunctions of equalities and negation of equalities on arithmetic expressions:

$$\Phi := true \mid false \mid E \ = \ E \mid E \ \neq \ E \mid \Phi \ \wedge \ \Phi$$

  An example of relational assertion: $y_1 = x_2 \wedge \Sigma_1[0] = \Sigma_2[0]$.

For a pair of states $(s_1, s_2) = (\langle pc, \sigma_1, \rho_1\rangle, \langle pc, \sigma_2, \rho_2\rangle)$, the semantics $[\![E]\!](s_1, s_2)$ of an arithmetic expression E, is inductively defined by:

$$\begin{array}{rclcrcl} [\![n]\!](s_1,s_2) & = & n & ; & [\![E+E']\!](s_1,s_2) & = & [\![E]\!](s_1,s_2) + [\![E']\!](s_1,s_2) \\ [\![x_1]\!](s_1,s_2) & = & \rho_1(x) & ; & [\![x_2]\!](s_1,s_2) & = & \rho_2(x) \\ [\![\Sigma_1[i]]\!](s_1,s_2) & = & \sigma_1[i] & ; & [\![\Sigma_2[i]]\!](s_1,s_2) & = & \sigma_2[i] \end{array}$$

$[\![\Phi]\!]$ denotes the semantics of a relational assertion Φ: it is a subset of $\mathbb{S} \times \mathbb{S}$ containing the couples of states (s_1, s_2) that satisfy Φ. $[\![\Phi]\!]$ is now defined as follows:

$$[\![true]\!] = \mathbb{S} \times \mathbb{S} \ ; \ [\![false]\!] = \emptyset \ ; \ [\![\Phi \wedge \Psi]\!] = [\![\Phi]\!] \cap [\![\Psi]\!]$$
$$[\![E \ = \ F]\!] = \{(s_1, s_2) | \ [\![E]\!](s_1, s_2) \ = \ [\![F]\!](s_1, s_2)\}$$
$$[\![E \ \neq \ F]\!] = \{(s_1, s_2) | \ [\![E]\!](s_1, s_2) \ \neq \ [\![F]\!](s_1, s_2)\}$$

For instance, our previous example $y_1 = x_2 \wedge \Sigma_1[0] = \Sigma_2[0]$ is satisfied when variable y in state 1 has the same value as x in state 2 and both stacks have the same top value.

We also use additional handy notations:

- for z a variable or a stack position, $\Phi[\backslash z]$ denotes relational assertion Φ where every clause where z occurs is removed:

$$true[\backslash z] \text{ is } true \ ; \ false[\backslash z] \text{ is } false \ ; \ (\Phi \wedge \Psi)[\backslash z] \text{ is } \Phi[\backslash z] \wedge \Psi[\backslash z]$$
$$(E = E')[\backslash z] \text{ is } true \text{ if } z \text{ appears in } E \text{ or } E', \text{ otherwise } (E = E');$$
$$(E \neq E')[\backslash z] \text{ is } true \text{ if } z \text{ appears in } E \text{ or } E', \text{ otherwise } (E \neq E')$$

- for E an arithmetic expression and y a variable or stack position, $\Phi[E\backslash y]$ denotes relational assertion Φ where every occurrence of y is replaced by E.

Finally, we introduce two functions $shift_1$ and $unshift_1$ adapted from other works of implementation of a program logic on bytecode [4] to enable easier manipulation of the stack in our relational assertions:

- $shift_1(\Phi) = \Phi[\Sigma_1[i+1]\backslash\Sigma_1[i]$ for all $i \in \mathbb{N}]$ is a global parallel substitution on the first stack variables replacing each position reference by the next one (as if an element was pushed on the stack);

- $unshift_1(\Phi) = (\Phi[\backslash\Sigma_1[0])\ [\Sigma_1[i]\backslash\Sigma_1[i+1]$ for all $i \in \mathbb{N}]$, all mentions of the top value of the first stack are suppressed, then a global parallel substitution on the first stack variable replaces each position reference by the previous one (as if an element was popped);

as well as similar definitions for $shift_2$ and $unshift_2$ (for the stack of the second program), $shift_{both}$ and $unshift_{both}$ (for both stacks).

2.2 Statements and their interpretation

Let C_1 and C_2 be two bytecode programs. A **statement** in RHL is a construction $\{\Phi\}\ pc \sim pc'\ \{\Psi\}$, with pc, pc' two program counter values for C_1 and C_2, and Φ, Ψ two binary relational assertions. Such statement can be read "if the precondition Φ stands when C_1 is in pc and C_2 in pc', then C_1 and C_2 behaviours are bound by postcondition Ψ". The exact meaning of this is correlated to the notion of valid RHL statement.

We say that a RHL statement is **valid** and write $\vDash \{\Phi\}\ C_1 \sim C_2\ \{\Psi\}$ if for any pair of states $(s_1, s_2) \in [\![\Phi]\!]$ satisfying the precondition Φ:

- either there is no state s_1' such that $s_1 \Rightarrow s_1'$ and no state s_2' such that $s_2 \Rightarrow s_2'$ (**both** executions diverge);
- either all pairs of states (s_1', s_2') such that $s_1 \Rightarrow s_1'$ and $s_2 \Rightarrow s_2'$ satisfy $(s_1', s_2') \in [\![\Psi]\!]$ (**both** executions terminate and satisfy the postcondition Ψ). Due do the determinism of our fragment of bytecode language, note that if both executions terminate there is a unique such (s_1', s_2').

2.3 Axiomatic semantics

Hoare logic over a programming language comes with a proof system of inference rules. In a similar fashion it is possible to build a system of inference rules for RHL statements. A statement is **provable** if it can be derived using the inference rules, in which case we write $\vdash \{\Phi\}\ pc \sim pc'\ \{\Psi\}$.

We present a sample inference rules system for our toy Java Bytecode programs. The system includes three types of rules: structural, one-sided and two-sided rules.

$$\text{FALSE}\quad \vdash \{\mathit{false}\}\ pc\ \sim\ pc'\ \{\Psi\} \qquad\qquad \text{CONS}\quad \frac{\vdash \{\Phi\}\ pc\ \sim\ pc'\ \{\Psi\}}{\vdash \{\Phi'\}\ pc\ \sim\ pc'\ \{\Psi'\}}\ [\![\Phi']\!] \subseteq [\![\Phi]\!] \text{ and } [\![\Psi]\!] \subseteq [\![\Psi']\!]$$

Figure 5: Structural rules for our RHL fragment.

Structural rules (Figure 5) can be applied regardless of the content of both programs. They involve logic manipulation of the precondition or the postcondition relational assertions:

- the rule $[FALSE]$ allows to prove any statement whose precondition is validated by no state;
- the rule $[CONS]$ (consequence) allows to strengthen the precondiction and/or weaken the postcondition of any statement. The way the rule is read is that if we have $\vdash \{\Phi\}\ pc\ \sim\ pc'\ \{\Psi\}$ (a proof for statement $\{\Phi\}\ C_1 \sim C_2\ \{\Psi\}$) then by using $[CONS]$ we can build a proof for $\{\Phi'\}\ C_1 \sim C_2\ \{\Psi'\}$ where Φ' is stronger than Φ and Ψ' is weaker than Ψ.

Note that $[CONS]$ is the only rule of our system that has **semantic requirements** rather than purely syntactic ones: as indicated on the right of the rule, it requires that some relational assertions are the consequences of others. $[CONS]$ is notably needed to modify the form of preconditions to match the ones required by other rules.

Other kinds of rule care about the instructions of the bytecodes. We use the notation $C.pc$ to refer to the instruction at program counter pc in program C. One-sided rules (Figure 6) account for an instruction on only one execution. They are declined in two versions, one for the left program and one for the right program (only left versions are shown on Figure 6). The effect of the instruction is reflected in the precondition in the premise. For instance:

$$\text{PUSH-L}\quad \frac{C_1.pc = push\ v \quad \vdash \{shift_1(\Phi) \wedge \Sigma_1[0] = v\}\ pc+1\ \sim\ pc'\ \{\Psi\}}{\vdash \{\Phi\}\ pc\ \sim\ pc'\ \{\Psi\}}$$

$$\text{LOAD-L}\quad \frac{C_1.pc = load\ x \quad \vdash \{shift_1(\Phi) \wedge \Sigma_1[0] = x_1\}\ pc+1\ \sim\ pc'\ \{\Psi\}}{\vdash \{\Phi\}\ pc\ \sim\ pc'\ \{\Psi\}}$$

$$\text{STORE-L}\quad \frac{C_1.pc = store\ x \quad \vdash \{unshift_1(\Phi[\backslash x_1][x_1 \backslash \Sigma_1[0]])\}\ pc+1\ \sim\ pc'\ \{\Psi\}}{\vdash \{\Phi\}\ pc\ \sim\ pc'\ \{\Psi\}}$$

Figure 6: One-sided left rules for our RHL fragment. The system implicitly contains a right (-R) version of all left (-L) rules.

- the rule $[PUSH - L]$ intuitively says that if there is a *push* instruction in C_1 at *pc* and if we have a proof for the statement after the *push* is done (stack is augmented with value v, *pc* is one step further), we can a deduce a proof of the statement before the push. $[LOAD - L]$ is very similar with a variable value instead of a constant;
- $[STORE - L]$ premise's precondition forgets all properties about x_1, since x_1 value was just modified by the *store*, replaces any mention of the stack top by mentions of x_1 and unshifts the left stack (since an element was popped).

Two-sided rules can be applied for pairs of programs of similar shape. A similar operation is executed on each side and both program counter advance. We included here:

- the rule $[END]$ which allows to conclude a proof if both programs have no instructions left and precondition is the same as postcondition;
- two-sided versions of the one-sided stack manipulation rules in case of similar stack manipulations on both sides: $[PUSH - BOTH]$, $[LOAD - BOTH]$ and $[STORE - BOTH]$;
- the rule $[IFEQ - BOTH]$ which accounts for conditional branching when we know that both sides will have the same branching behaviour. The premise requires a proof that the statement holds no matter what this branching behaviour is.

An example proof of a RHL statement about the programs of Figure 3 can be found on Annex 47.

Note that we did not include a one-sided rule for conditional branching in our system. This means we cannot handle pairs of programs with different conditional control flow. The rule system is not **complete**: there are some valid RHL statements that are not provable.

A work of Benton [8] proposes such a system of rules and shows that it is **sound**: any provable RHL statement is valid. However the use of RHL for equivalence verification is not trivial: Benton's system is not complete either and proving a RHL statement requires clever use of the $[CONS]$ rule. The next section reviews proof strategies based on such relational approaches.

3 Relational program verifiers

Relational Hoare logic is one example of a relational tool that could be implemented in a program equivalence checker. In this section we review equivalence verifiers implementing such relational frameworks.

$$\text{END}\quad \frac{C_1.pc = return \quad C_2.pc' = return}{\vdash \{\Phi\}\ pc\ \sim\ pc'\ \{\Phi\}}$$

$$\text{PUSH-BOTH}\quad \frac{C_1.pc = push\ v \quad C_2.pc' = push\ v \quad \vdash \{shift_{both}(\Phi) \wedge \Sigma_1[0] = \Sigma_2[0]\}\ pc+1\ \sim\ pc'+1\ \{\Psi\}}{\vdash \{\Phi\}\ pc\ \sim\ pc'\ \{\Psi\}}$$

$$\text{LOAD-BOTH}\quad \frac{C_1.pc = load\ x \quad C_2.pc' = load\ y \quad \vdash \{shift_{both}(\Phi \wedge x_1 = y_2) \wedge \Sigma_1[0] = \Sigma_2[0]\}\ pc+1\ \sim\ pc'+1\ \{\Psi\}}{\vdash \{\Phi \wedge x_1 = y_2\}\ pc\ \sim\ pc'\ \{\Psi\}}$$

$$\text{STORE-BOTH}\quad \frac{C_1.pc = store\ x \quad C_2.pc' = store\ y \quad \vdash \{unshift_{both}(\Phi[\backslash x_1, y_2][x_1 \backslash \Sigma_1[0], y_2 \backslash \Sigma_2[0]]) \wedge x_1 = y_2\}\ pc+1\ \sim\ pc'+1\ \{\Psi\}}{\vdash \{\Phi \wedge \Sigma_1[0] = \Sigma_2[0]\}\ pc\ \sim\ pc'\ \{\Psi\}}$$

$$\text{IFEQ-BOTH}\quad \frac{\begin{array}{c}C_1.pc = ifeq\ l \quad C_2.pc' = ifeq\ l' \\ \vdash \{unshift^2_{both}(\Phi[\backslash \Sigma_1[0], \Sigma_1[1], \Sigma_2[0], \Sigma_2[1]])\}\ pc+1\ \sim\ pc'+1\ \{\Psi\} \\ \vdash \{unshift^2_{both}(\Phi[\backslash \Sigma_1[0], \Sigma_1[1], \Sigma_2[0], \Sigma_2[1]])\}\ l\ \sim\ l'\ \{\Psi\}\end{array}}{\vdash \{\Phi \wedge \Sigma_1[0] = \Sigma_2[0] \wedge \Sigma_1[1] = \Sigma_2[1]\}\ pc\ \sim\ pc'\ \{\Psi\}}$$

Figure 7: Two-sided rules for our RHL fragment.

3.1 Semantic differential for PHP language

Benton built an automated semantic difference finder [8] in order to compare the code produced by two compilers for Hack, a typed PHP variant. The semantic core of the tool uses relational Hoare logic and performs a semantic comparison of the instructions at intraprocedural level for functions of the same name.

As mentioned before, the RHL checker is shown to be sound but not complete: if it outputs that the two programs are equivalent then they really are; however it might report a difference when there actually is no semantic difference. In case it detects a difference, the tool highlights the instructions where this discrepancy is detected for human eyes to decide if this potential difference is real. If the program bits are actually equivalent, the tool can be adapted by strengthening the RHL core to catch this equivalence or by adding an ad-hoc pattern for equivalence on top of the logic analysis.

Since Hack is also based on a stack bytecode language and Benton's goal is quite similar to our, we took inspiration from his framework to describe RHL in section 3. There are a few key differences between what we present and his approach:

- RHL statements all have the same implicit postcondition relation: both program have **the same observable behaviours** (termination behaviour + input/output actions);
- precondition relational assertions are conjunctions of local variable equalities that implicitly require that states have **equal stack content**. Note that requiring equal stack content is very restrictive and only works here because the two programs are expected to be very similar.

Benton highlights practicalities and limits of proof strategies using RHL. For example one-sided rules in the axiomatic system introduce non-determinism in choosing which rule should be used: in his case, Benton decides to do as many one-sided steps as possible. Such an heuristic is needed to avoid systematic exploration of the full space of possible rule choices but may lead to struggling on pairs of bytecodes of different control flows: since we must use one sided rules in that case, the heuristic is very important.

Another limit of this technique for our case is that it worked well because Benton could modify the Hack compiler: whenever a translation quirk was hard to capture with RHL, they could improve the tool but also simply patch it in the compilers. We cannot do so for the Kotlin compiler.

3.2 Program product analysis

In their book [6], Barthe compares Relational Hoare logic with another common method for establishing relational properties between programs: the construction of a program product. The gist of this technique is to express relational properties about the two programs we want to compare as invariants of a single **program product**.

A specific set of instructions is deemed to be critical in the two programs: those are the instructions around which the two programs' behaviours will be compared. Typical critical instructions may include input/output (*return*) and control flow manipulations (*if*). The other instructions are considered non-critical.

A program product contains all non-critical instructions for both initial programs, manipulating independent set of variables. Critical instructions are merged and the new program is decorated with assumptions and assertion establishing a desired link between the two set of variables. Figure 8 is an example of program product for two very basic programs. Note that since they manipulate independent variables, the ordering of the non-critical instructions of the two programs between two critical instructions is unimportant. However a difficulty lies in this construction since it may involve non-determinism for pairing the critical instructions together. This task is simplified if the two programs have a very similar structure.

```
//Program 1
y = 2 × x
z = x + 1
return z
```

```
//Program 2
y = 2 × x
z = y - x + 1
return z
```

```
//Program product
assume x1 = x2
y1 = 2 × x1
z1 = x1 + 1
y2 = 2 × x2
z2 = y2 - x2 + 1
assert z1 = z2
return z1
```

Figure 8: A program product of two pseudocode programs. In the program product two set of independent variables are used, each referring to the variables of initial programs (denoted by subscripts 1 and 2). We work under the assumption that both programs are given the same value for variable x. The *return* instructions are critical, we want to ensure that both program return the same value.

A program product is *valid* if all possible executions satisfy every assertion. The overall process (construction of the program product+validity check) is correct if the validity of the program product implies the desired property on the two initial programs (for example equivalence).

An example of program product analysis can be found in the work of Barthe et al. [5]. They want to check a property of relative-safety between two programs (if the first program is safe then the second program is too). They build a program product, then use a weakest-precondition calculus and a verification condition generator [12] they implemented to check its validity.

The RHL approach and the program product construction are shown to be linked by Barthe [6]: a relational Hoare logic statement over two program is valid if and only if the same statement interpreted as Hoare logic statement over the program product is valid.

Both examples of relational verifiers we reviewed did implement their own specialized logic solvers to discharge ad-hoc proof obligations. In the next section we study an equational approach that transcribes our problem in a form that allows the use of generic solvers.

4 Advanced decision procedures for term equivalence

Congruence closure [9] is a problem over terms of uninterpreted function symbols. We want to know if two terms can be made equal from some already known equalities and the principle of congruence (which essentially states "if a equivalent to b then for all function symbol f, $f(a)$ equivalent to $f(b)$"). For example, if we know that $a \equiv f(a)$, we can identify $g(f(a))$ and $g(f(f(a)))$ by congruence. The following inference rule is an inductive definition of congruence; an equivalence relation is a **congruence relation** if it is closed under this inference rule:

$$\text{CONG}\quad \frac{a_1 \equiv b_1 \quad \dots \quad a_n \equiv b_n}{f(a_1, \dots, a_n) \equiv f(b_1, \dots, b_n)}$$

The problem of congruence closure amounts to finding the smallest equivalence relation that contains a set of known equivalences and is closed under this inference rule. This framework allows to build an efficient decision procedure for program equivalence. In the rest of this section, we first study the elements powering this decision procedure in the general context of uninterpreted terms, then we review some articles that specialize this method for programs in Subsection 4.4.

4.1 E-graphs

Before getting into the algorithms, we present a data structure adapted for working on equivalence classes of terms. We want a data structure that will represent and support operations for both equivalence relations and subterm relations hence why we cannot simply use union-find.

Equality graph (e-graph) is an efficient representation of a congruence relation over terms. Historically e-graphs were developed to represent known terms and equalities over them in automated provers; for example, e-graphs are used in some SMT solvers. We present the general definition of e-graphs given by a recent paper from Willsey et. al. [18] that implements them in a library named *egg*.

At its core, an e-graph is a variant of a term graph with sharing: nodes represent terms and contain a function symbol; a node representing the term $f(a_1, ..., a_k)$ will have ordered edges pointing towards the nodes a_1, ..., a_k. The main difference with term graph is that in an e-graph, the edges no longer point directly towards a term node but towards an equivalence class (**e-class**) of nodes (**e-nodes**). This means that an e-node no longer represents a single term $f(a_1, ..., a_k)$ but possibly many equivalent terms obtained by replacing any subterms a_i by equivalent subterms. Sharing is extended from equal terms to equivalent terms.

Figure 9 shows some e-graphs for arithmetic expressions. Subfigure (a) is an e-graph that represents the term $(a \ll 1)/2$ ($\ll$ is the bitwise left shift operation): its essentially the term graph of this expression, except edges point towards one-element e-classes containing each e-node. Subfigure (b) is a similar e-graph for term $(a \times 2)/2$: there is sharing for the two occurrences of 2. E-graph (c) can be seen as the merging of e-graphs (a) and (b) when making terms $(a \times 2)$ and $(a \ll 1)$ equivalent: their e-nodes end up in the same e-class and the top /-node now represents two different terms.

In *egg*, the e-classes of an e-graph are sets of e-nodes and are given an unique integer identifier (id) for immutable representation. An e-node is a pair of a symbol function and a list of children e-classes ids. The e-graph structure is entirely described by a union-find on e-class ids and a map of e-class ids to actual e-classes. Some additional information is also stored to improve the efficiency of data structure operations: for instance, a system of hashconsing from e-nodes to e-class ids allows for quick presence/e-class membership check of an e-node.

E-graphs support the two basic union-find operations on e-classes ids:

- *find*, which takes an e-class id a and returns the canonical id associated to a in the union-find structure;

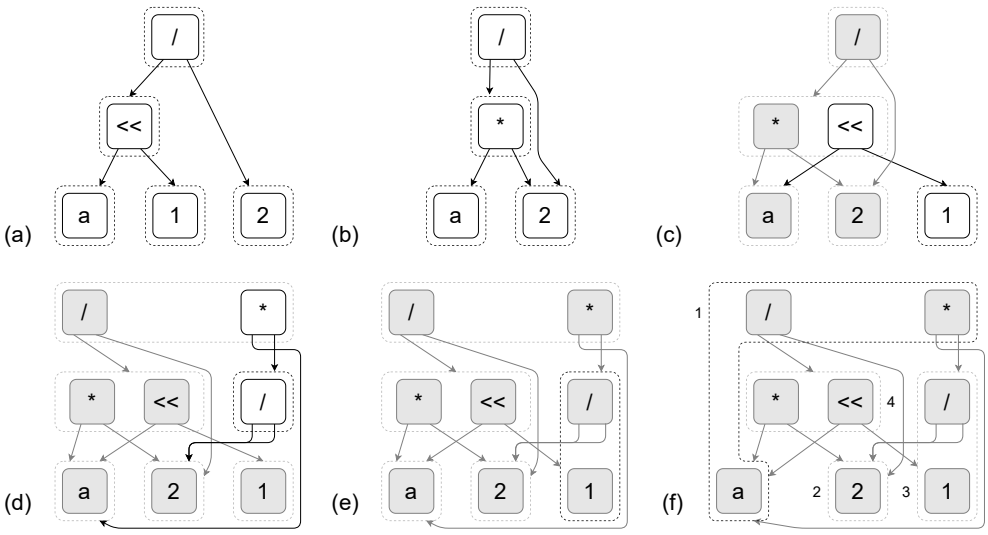


Figure 9: Examples of e-graphs for arithmetic expressions. E-classes are circled by dotted lines, and given identifiers on Subfigure (f) for easier referencing. These examples are used thorough Section 4.

- *union*, which takes two e-classes ids a and b and merges the corresponding equivalence classes in the union-find structure.

Two supplementary operations are offered: *ematch* and *add*. We delay the presentation of *ematch* until Subsection 4.3. *add* takes an e-node description n and determines if n is already in the e-graph. If n is found in the e-graph, *add* returns the id of the e-class of n. Otherwise it creates a new e-class containing only e-node n and returns its id.

The *add* and *union* operations allow to enrich an e-graph with new information. For instance on Figure 9, e-graph (c) can be obtained from e-graph b by adding two new e-nodes (one for 1, one for $(a \ll 1)$) and merging the e-classes of the $\times$ e-node and the $\ll$ e-node.

Allowing modifications of the content of the data structure mean that the equivalence relation represented by an e-graph might not be a congruence relation anymore after the use of these operations. The next subsection goes over methods to preserve a congruence invariant.

4.2 E-graphs and congruence relations

The problem of finding the smallest congruence relation that contains an equivalence relation (its **congruence closure**) has historically been studied since the 1970s. Downey et al. introduced algorithms to compute congruence closure as a relation over the vertices of an oriented graph [9]. This context is more general than our problem, since the notion has applications not only for term equivalence but also to check if a join is lossless in a database given a set of functional dependencies between the attributes.

The technique used in *egg* [18] to enforce the congruence invariant on e-graphs is a rebuilding procedure similar to the algorithms described to the algorithms of Downey et al: it is an upward traversal of the e-graph, merging the e-classes predecessors of the merged e-class that have become congruent. Namely:

- look at the parent e-nodes of the merged e-class. If any two are now congruent (i.e. have the same representation symbol function+children e-class id), merge their e-classes;
- call back the procedure on any new merged class created by the previous step.

Figure 10 presents an instance of this algorithm on a basic e-graph.

One of the contributions of *egg* is noticing that these rebuilding calls can sometimes be delayed and done in a batch after several merges: they chose to decouple the rebuilding procedure call from the merge

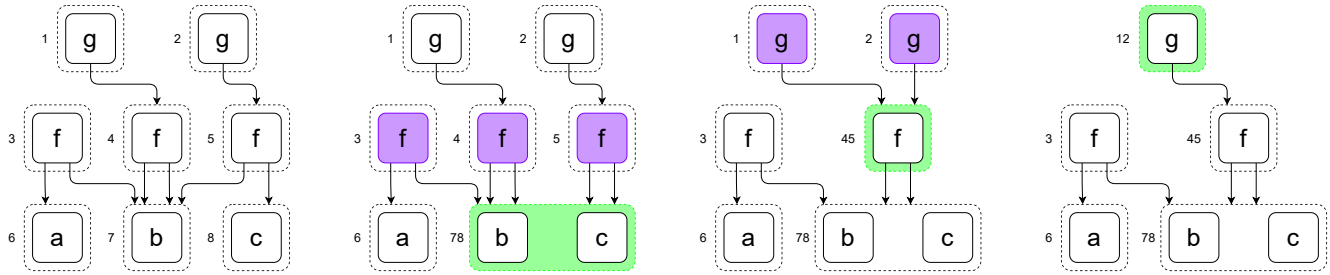


Figure 10: Example of the upward merge algorithm for enforcing the congruence invariant. Parent e-nodes of a new merged e-class are checked for equivalence. If two of them are equivalent their e-classes are merged and the rebuilding procedure must be recursively called.

operation and to give the responsibility of maintaining the congruence invariant to the user. Such a refactoring leads to overall better performances notably because it allows for new opportunity of parallelization in some cases, and some congruence closure calculations can be shared among all delayed rebuildings.

4.3 Equality saturation

The e-graph data structure offers an efficient framework for manipulation of congruence relations on terms. This in turns powers a new technique to saturate a set of terms with equivalent rewriting under some equivalence rules: equality saturation.

In the work of Tate et. al. [17], **equality saturation** is presented in the context of program optimization as an alternative to classical rewriting approaches. From an initial program P and a set of rewriting rules, we want to find an optimized equivalent form of P.

Classical rewriting methods are destructive, in the sense that they transform the program by successively applying the rewriting rules. The main problem these approaches face is called phase ordering: the order in which the transformations are applied to the program matters. Many different sequences of rewrites are possible and they do not lead to the same level of optimization (if any).

An additive approach that keeps track of all forms of the program seen and only adds new information about equivalences of terms whenever a rewriting rule is applied would solve this problem at the cost of performance if implemented naively. To ensure efficient rewriting, equality saturation takes this additive approach and exploits the sharing properties and powerful operations offered by e-graphs, notably the *ematch* querying operation.

ematch takes as input a pattern term p with variable placeholders and returns the list of all occurrences of the pattern p in the e-graph. An occurrence of p is described by a couple (σ, a) where σ is a map from the variables of p to e-class ids and a the id of the e-class where $\sigma(p)$ was found. Essentially, *ematch* returns a list of e-classes that contain the desired pattern and the substitutions of the pattern variables needed to identify the pattern.

As an example, consider the e-graph (f) of Figure 9. Searching for the pattern $(x/2)$ with x a variable placeholder would return two occurrences: one in e-class 1 with e-class 4 as value for x and another one in e-class 3 with e-class 2 as value for x.

The general workflow of equality saturation is described in Figure 11. It takes as input some terms and a set of rewriting rules: the leftmost part of a rule is a term pattern with variables, the rightmost part is a resulting equivalent term over the variables of the pattern. The procedure is as follow:

- start from an e-graph representing the terms we want to study;
- until satisfied, apply all possible rewriting rules: find an occurrence of the pattern of a rewriting rule by e-matching and add to that e-node's e-class the term defined by the rightmost part of the rule;

- stop when satisfied: either the e-graph is saturated (no more rewriting rule can be applied), or some timeout condition is reached.

```
def equality_saturation(term, rewrites):
    egraph = initial_egraph(term)
    while not saturated or timeout:
        for (pattern,result) in rewrites:
            for (subst, eclass) in egraph.ematch(pattern):
                eclass2 = egraph.add(subst(result))
                egraph.union(eclass,eclass2)
```

Figure 11: Pseudocode for equality saturation.

The e-graphs (b) to (f) of Figure 9 can be seen as an example of progressive saturation of the e-graph of $(a \times 2)/2$ by applying successively some rewriting rules. From (b) to (c), a rule $x/2 = x \ll 1$ is used. From (c) to (d) it would be $(x \times y)/z = x \times (y/z)$. From (d) to (e), the rule is $x/x = 1$; note that applying this rewriting rule does not add any new e-node, it simply leads to two e-classes being merged. From (e) to (f), it is $x \times 1 = x$. The e-graph (f) is saturated for these rules. In this example, we only applied each rule once, in general a rule might be applied in several different e-classes of the e-graph and the applying of subsequent rules may create new occurrences of the pattern.

The final e-graph (f) shows that the initial expression $(a \times 2)/2$ is equivalent to just a (both e-nodes are in e-class 1); it is interesting to note that the first rewriting rule application $x \times 2 = x \ll 1$ did not help achieving that result. A term rewriter based on a destructive approach could have been stuck with the bitshift expression $(a \ll 1)/2$ but equality saturation did not lose the information about the initial term so it can still apply the other rules to it.

In the context of term rewriting, the final e-graph is scanned in search for an optimal representing e-node for each e-class according to a given optimization criterion. For the purposes of program equivalence checking, the procedure can be stopped if the e-classes corresponding to the two programs are merged, in which case we can report semantic equivalence.

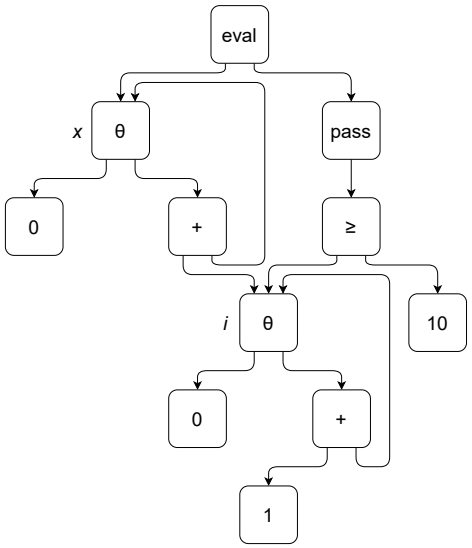
4.4 Congruence closure for program manipulation

Congruence closure and equality saturation can lead to a powerful decision procedure for terms equivalence. If we define a term representation of a program and rewriting rules for equivalence over those terms, we will be able to use the algorithmic core of equality saturation for our problem of program equivalence.

Stepp, Tate et al. built a tool named Peggy based on equality saturation [15, 17]. The Peggy framework is used to perform the various applications of equality saturation on programs: optimizing rewriting of Java bytecode or translation validation over the optimizations done by LLVM. They give a formal semantics of their representation of programs, a few examples for pseudo-code and the rewriting rules they use.

The intermediate representation of programs they use is called E-PEG (Program Expression Graph). It's essentially a variant of e-graphs specialized for their term language. The E-PEG of a program is computed from its control flow graph. This means that the E-PEG conversion can already capture some program equivalences due to modifications that do not alter the control flow graph (code placement for instance).

In order to translate an E-PEG from or to a CFG, terms must be interpreted. They can take values of different types: integer, boolean, domain specific types of the target language, $\perp$ (undefined value, for programs that do not terminate for instance). Some terms (notably those using the function symbol θ) are interpreted as sequences of values of those basic types: this is useful for loop encoding.




```
1 x = 0
2 while (i<=10){
3     x+=i
4     i+=1
5 }
6 return x
```

Figure 12: An example of term associated to a simple pseudocode. We annotated the θ-terms with the variables they correspond to in the pseudo-code. Note that the graph is cyclic since loop sequence values are computed from the previous value of the sequence.

There are two types of function symbols: primitive functions that allow encoding of the control flow graph, and domain-specific functions (such as arithmetic) that will depend on the language fragment we want to study. The four primitive function symbols are interpreted as follows:

- $\phi(cond, if, else)$, inspired by the static single assignment-intermediate representation, denotes a conditional branching (*if then else*) in the CFG. *cond* is interpreted as the condition for branching, *if* as the value if *cond* is true, $else$ as the value if *cond* is false;
- $\theta(base, loop)$ is interpreted as a sequence of values taken over a loop. *base* is the head value of the sequence, and *loop* is its tail sequence;
- $eval(loop, ind)$ takes a sequence of values *loop* and an integer *ind* and returns the value of *loop* at index *ind*;
- $pass(seq)$ takes a sequence of booleans *seq* and returns the first index (if it exists) at which *seq* value is true.

Figure 12 presents a simple pseudocode program with a loop and the associated term.

The authors also give some built-in equivalence rules for their primitive function symbols, for instance $pass(\theta(true, x)) \equiv 0$ (the first index where true is met in a sequence starting by true is 0). To power the Peggy tool, the list must be enriched with domain specific equivalence rules.

This base term language can be extended to take into account more programming language operations. For instance, if the program modifies the memory, terms are now allowed to be interpreted as an heap state. The CFG of a program will be represented by two terms rather than one: the first one is interpreted as the return value as we presented, the other one as the heap state after execution of the program. The symbol functions is then extended with primitives for interacting with memory such as *load* and *store*.

Stepp and Tate illustrate how the equational approach on terms can be adapted to programs. This decision procedure is much more generic than the specialized relational verifiers of Section 3. The semantics

of EPEGs is however quite complex. While the approach is sound, it is harder to build proofs of soundness for it than it is for relational program verifiers. This is a caveat of this technique we will have to keep in mind.

5 Conclusion

As program equivalence is undecidable in general, our verifier must be tailored to our specific context of Java to Kotlin translation validation. The first step of the internship will be to identify the algorithmic level needed to establish equivalence between the bytecodes. We may need a middle ground between the specific analyses of relational program verifiers and the genericity of equational decision procedures, or we may have to use other static analysis techniques.

Part II
Internship report

6 A differential of Java and Kotlin programs

We dedicated a first part of the internship to a survey of the differences generated by the Java to Kotlin conversion and the compilers. We had two main goals:

- discover some specific differences introduced by the languages specificities and the two different compilers;
- assess the general difficulty of the equivalence problem in this specific case, and pick a technique from the bibliography accordingly.

This part of the internship was very exploratory. This section aims to present our general experimental framework, some of the tools we used and the observations we made.

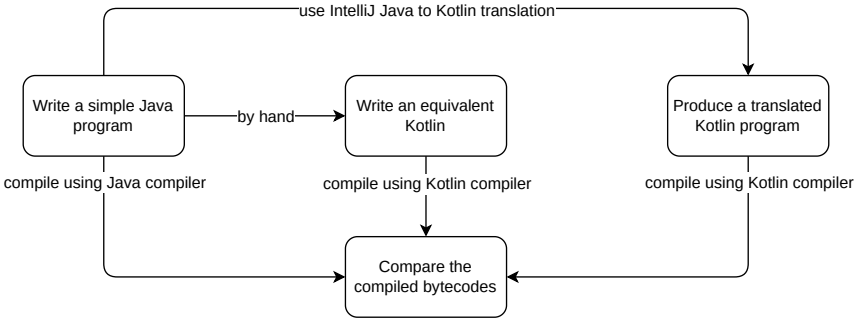


Figure 13: Schematic representation of one experience.

An important aspect of our work was that we did not have access to the code of an automated Java to Kotlin translator. The simplest course of action for our survey was to study a variety of programs as we translate and compile them, using the compilers and translator as opaque boxes. One of our experiments can therefore be divided in the four steps summarised on Figure 13:

- we write a very short Java class that targets one specific feature of the Java language.
- we write an equivalent Kotlin class. The internship is mainly motivated by the case of an automatic translation of Java to Kotlin but the case of programmer doing the translation by themselves is also interesting.

- we use the IntelliJ Java to Kotlin automatic translation to convert our Java class to a Kotlin class. This sometimes produces the same program as the by-hand method but the translator tends to use more idiomatic constructions of Kotlin.
- we compile the three classes and compare the obtained bytecodes. We are mostly interested in the comparison between the Java-originated bytecode and any of the two Kotlin-originated bytecode.

We also used the Peggy equivalence checker from the bibliographic study [17] and confronted its completeness to our specific problem. The tool is available online and managed to instantly prove equivalence on some of our non-trivial examples with different control flows in the Java and the Kotlin program. No other such tool implementing a relational verification approach from the bibliography was available. We decided to focus the internship on the equality saturation technique and specific kinds of programs that made Peggy struggle.

Studying bytecode is very tedious so we only present pseudo-code in this section. This is also the only part of the internship during which we interact directly with bytecodes: starting from Section 7 we will work on Peggy intermediate representation instead.

Section 6.1 presents the main feature differences between the Java and the Kotlin languages that guided our experiments. Sections 6.2 and 6.3 present both tools we used in this survey, the IntelliJ converter and Peggy. Section 6.4 presents some of our studied programs and the observed differences between bytecodes.

6.1 Java and Kotlin main differences

Although our survey aims to range over a variety of basic programs, we are guided by existing differences between the Java language and the Kotlin language. The Kotlin documentation [2] contains a list of features that differ between the two languages. We mainly studied two features.

The first one is relatively minor but was consistently found in our basic programs: the if-structure in Kotlin is an expression, akin to the much rarer ternary operator $a?b : c$ in Java. In Kotlin, it is common and idiomatic to return an if-expression when relevant. Since the most basic programs we could conceive were usually variations of if-then-else structures, this idiomatic if-expression appears very often in the corresponding Kotlin programs.

The other difference is much more major and a main selling point of Kotlin: the type system is responsible for nullness safety. Each reference type such as *String* has a nullable variant *String?* for strings that might also be *null*. *String* is the type of string references that are known not to be *null*.

In Kotlin, calling a method with a nullable reference such as a *String?* as receiver requires first checking that the reference is not *null*, for example in a condition. The Kotlin compiler is able to detect these checks and then allows the call. The user can also use a variety of built-in operators like the safe call operator *?* (a call received by *s?* such as *s?.count()* returns *null* if *s* is *null* and the result of the call otherwise).

There is an operator called the not-null assertion operator *!!* which mimics the Java behaviour of throwing a Null Pointer Exception if the reference we call upon is *null*. This operator appears often in our context of automated translation from Java (about every time a potentially unsafe call is made in the Java program).

These feature differences between Java and Kotlin are a source of differences between the bytecodes. There are other sources such as the compilers themselves and the translation of Java to Kotlin. In the next subsection we present the automated translator we studied during the internship.

6.2 The IntelliJ Java to Kotlin translator

IntelliJ IDEA is an IDE released by JetBrains [1] that focuses on Java and Kotlin development. Among its functionalities, it includes a Java to Kotlin converter. This is the example of automated translator we studied during the internship.

The program produced by the converter is advertised as "not supposed to be perfect, however, it may be helpful in some situations". We observed the following interesting behaviours:

- the conversion is intraprocedural and "overall line-by-line": one Java function becomes one equivalent Kotlin function, and one basic step or structure of the Java program is converted into one expression in the Kotlin code;
- the converter transforms some structures of the Java program into structures more idiomatic to Kotlin (such as expression-if) when it can;
- the null safety feature of Kotlin is adequately dealt with (this requires more than a line-by-line view of the program). The converter correctly adds defensive checks to the produced Kotlin class to mimic the Java class behaviour in case of Null Pointer Exception;
- some small optimizations are performed on the produced code such as removing an intermediate assignment that only has one use;
- the converter generally preserves code comments but we could trick it into removing one by putting it on the same line as a useless intermediate assignment. This does not really matter in our study (since we care about the compiled program) but this is a somewhat interesting fact.

These translation quirks, as well as the compilation process itself sometimes introduces light differences in otherwise similar bytecodes. The comparison process of the bytecodes themselves can become somewhat tedious. The next subsection gives a general overview of the Peggy tool and how we used it in this phase of the internship.

6.3 The Peggy equality saturation tool

We very lightly discussed the Peggy tool [17] in the bibliography study. In this section we give a high level overview of the tool, its interface and how we used it in the exploratory part of the internship. Peggy became the main object of study of the internship but we delay to later sections the presentation of some of its other specific aspects.

Peggy is a research prototype and is notably the product of the PhD theses of Ross Tate and Micheal Stepp [16, 14]. It is a Java implementation of equality saturation for two main applications: optimization and proof of equivalence of Java or LLVM programs. The Peggy executable and its sources are available online [3]. As a sidenote, the project has not been maintained since 2011. Some light tinkering with the source code was required to allow Peggy to be used in year 2023 but we managed to get most of its functions to work (luckily, everything but the optimization engine, which we did not need).

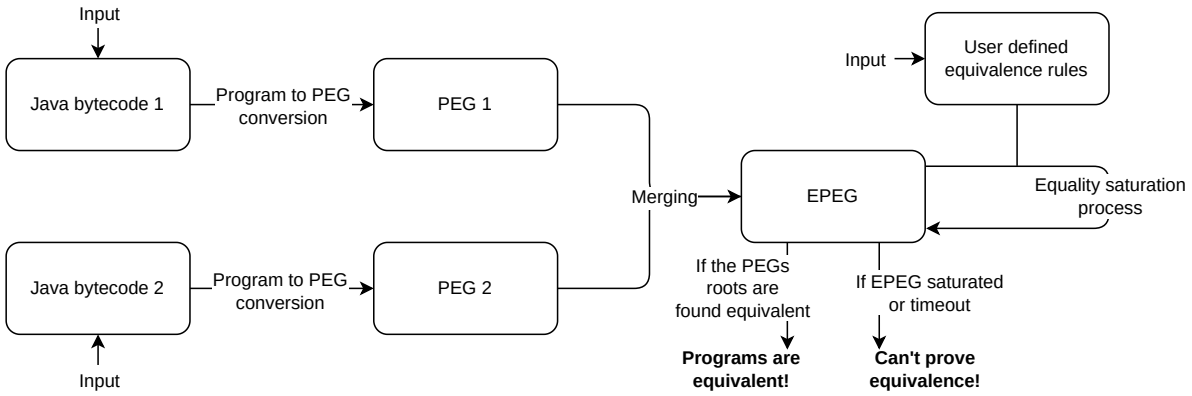


Figure 14: High-level overview of how Peggy performs equivalence checking.

In our case we mostly care about the equivalence checker, and only for Java bytecodes. The general concept of the equivalence checker of Peggy is summarised on Figure 14:

```java
int z;
if (x == 108) {
    z = x+60;
} else {
    z = x+50;
}
return z;
```

```kotlin
val z: Int
z = if (x == 108) {
    x + 60
} else {
    x + 50
}
return z
```

Figure 15: Basic Java code and associated translated Kotlin code. The if-structure in Java is translated to an expression-if in Kotlin.

- the user inputs two Java bytecodes for classes. They are turned into PEGs (Program Expression Graph), a term representation of their control flow graphs;
- the PEGs are merged into an EPEG (Equality PEG), a representation of equivalence classes of terms;
- the EPEG is enriched by an equality saturation process with more and more information about equivalences based on some equivalence rules. A base set of equivalence rules is proposed but more can be added to adapt the engine to specific cases ;
- if at some point the initial PEGs are shown to be equivalent in the EPEG, Peggy has proved equivalence. Conversely if no more rule can applied or a timeout condition is met without the PEGs being equivalent, Peggy reports not having been able to prove equivalence. A weakness of Peggy, and solver-like approaches in general, is that it is sometimes hard to explain why the proof failed: a timeout can be due to a problem of performances or simply an insufficient equivalence rules set.

Equality saturation was discussed in the bibliography study in Section 4.3: it essentially amounts to algorithmic techniques and clever data structures that allow to enrich a set of known equivalences between terms by applying some equivalence rules. PEGs and EPEGs are more thoroughly studied in Section 7. In Section 8 we try to use Peggy for our study case and discuss more in details the kind of equivalence rules needed.

6.4 Survey of basic programs

A listing of experiments and results can be found in Figure 48 (in Annex). For each feature we studied, we report a brief summary of the observed differences between bytecodes and whether or not Peggy was able to prove equivalence. In this section, we comment on some of those results.

It is extremely rare for the Java bytecode and the Kotlin bytecode to be exactly the same: at the the very least, there will be a permutation on used registers or the conversion process will lead to some useless code suppression (storing a local right before the function end for example). A strength of Peggy is that these bytecode-level differences are abstracted by its intermediate PEG representation. Many times during experiments, Peggy is able to report equivalence without even powering on the equality saturation engine since both programs are already associated with the same PEG.

A common example of this is by the expression-if of Kotlin, as shown in Figure 15. In the bytecode from Java, the computed value of z is stored in a register in each branch of the if, while in the Kotlin bytecode there is only one store instruction, after the if structure. In terms of high-level control flow, this result in the exact same graph.

Another interesting case that is covered by Peggy is the dead code suppression sometimes performed by the Kotlin compiler: if a conditional branching can be solved at compilation time by constant folding then one of the branches of the conditional is dead code and the Kotlin compiler only includes the alive

branch in the bytecode. The Java compiler does not have this behaviour and this leads to different control flows in the bytecodes. Peggy can perform constant folding and solve the conditional too, therefore proving equivalence.

Peggy struggled on three different cases. The first one is non-terminating programs. For an eternal while-loop program, the Kotlin compiler does not even include a return instruction at the end of the compiled bytecode. Peggy is not able to convert this bytecode into a PEG. The creators of Peggy discuss ways to deal with non terminating programs [16] but these features were not implemented so we did not explore too much in this direction.

Another difference is the fact that getting and setting the field of an object is done via a special instruction in Java bytecodes, while Kotlin classes are compiled with getters and setters as methods. Peggy does not know that a call to these methods is the same as accessing or setting a field.

It is interesting to note that we did not find anything more challenging than this difference in the object sphere of the languages. However we only studied very atomic programs and more subtle differences could appear due to the Kotlin nullness safety in more sophisticated programs.

Lastly, the most interesting difference we discovered is that a certain kind of for-loops on integers is not compiled the same way in Java and Kotlin. The pattern for these for-loops is an iterator

$$for(int\, i = i_0;\ i \leq i_f;\ i++)$$

that satisfies exactly these criteria:

- the iterator must be on *int*;
- the end condition must be a non-strict inequality $\leq$;
- the iteration must be an increment of one and use the ++ operator ($i = i + 1$ does not fit the pattern).

As shown on Figure 16, the Java compiler treats these structures as a standard while loop with *going over* i_f as an end condition. The Kotlin compiler on the other hand first checks if the initial value i_0 is under i_f then builds some kind of do-while loop with *reaching* i_f as an end condition.

```
1 #in Java
2 i = i0
3 while (i <= ifinal) {
4     loop body
5     increment i
6 }
```

```
1 #in Kotlin
2 i = i0
3 if (i <= ifinal) {
4     loop body
5     while (i != ifinal){
6         increment i
7         loop body
8     }
9 }
```

Figure 16: Pseudo-codes for the two different ways the $for(int\, i = i_0;\ i \leq i_f;\ i++)$ structure is compiled.

Those two control flows are very different, but equivalent. However, Peggy is unable to prove equivalence with its base set of equivalence rules. In order to solve this, we study a few programs containing this problematic loop-for structure, and how Peggy deals with them in Section 8. Since this study is heavily based on the PEG and EPEG structure, we propose an in depth description of the concept in Section 7.

7 Background: the EPEG program representation

In this section, we present the PEG and EPEG program representations, from the works of Tate et al [16, 17, 14], using the standardized vocabulary of the e-graph data structure [18]. Both the syntax and

the semantics of these representations are made more complex by nested loops in the programs. For pedagogical reasons, we propose a **first simplification:** we do not show any nested loop structures in this report so we present a *simplified* version of the syntax and semantics for programs **with no nested loops**. Throughout this document, our notations are harmonized and follow the table in Figure 17.

integers	$\mathbb{N}$	$\ni$	$i \mid j \mid k \mid ...$
function symbols for our terms	$\mathcal{F}$	$\ni$	$f \mid g \mid h \mid ...$
variable placeholders for terms	$\mathcal{V}$	$\ni$	$x \mid y \mid z \mid c \mid d \mid ...$
terms	$\mathcal{T}$	$\ni$	$t \mid t' \mid ...$
patterns	$\mathcal{P}$	$\ni$	$P \mid P' \mid ...$
substitutions	$\mathcal{S}$	$\ni$	$\sigma \mid \sigma' \mid ...$
loop identifiers	$\mathcal{L}$	$\ni$	$l \mid l' \mid ...$
loop iteration indices	$\mathbb{I}$	$\ni$	$\iota \mid \iota' \mid ...$
e-nodes	$\mathcal{N}$	$\ni$	$m \mid n \mid n' \mid ...$
edges	$\mathcal{E}$	$\ni$	$e \mid e' \mid ...$
paths	Π	$\ni$	$\pi \mid \pi' \mid ...$
e-classes	$\mathcal{C}$	$\ni$	$A \mid B \mid W \mid X \mid Y \mid Z \mid ...$
e-classes of boolean values	$\mathcal{B}$	$\ni$	$C \mid D \mid E \mid ...$
sets of boolean classes (contexts)	$2^{\mathcal{B}}$	$\ni$	$\chi \mid \chi' \mid ...$

Figure 17: Symbol table for the concepts used in this document.

7.1 Syntax of PEGs and EPEGs

The PEG and EPEG data structures represent terms over a specific term language $\mathcal{F}$. This language contains domain function symbols (arithmetic operations, logic operations, input of the program, etc.) and some specific function symbols:

- Φ is a ternary function symbol that represents conditionals;
- θ is a binary function symbol that represents sequences of values (a variable in a loop for example);
- *pass* is an unary function symbol that represents exit condition for loops;
- *eval* is a binary function symbol that represents loops.

We give a more precise description of the meaning of these symbols in Section 7.2.

Terms over those symbols are allowed to be cyclical in some given ways: there is a well-formedness criterion for allowed cycles, but we do not go over it; all terms in this report are well-formed. An example of such a term is given in Figure 18 *(a)*. We regularly use the intermediary definition technique to write a cyclical term as we did for the loop on the θ symbol. y is a function symbol for an input of the program. A PEG is a graph representation of such a term:

Definition 1 (Program Expression Graph)**.** *A* ***Program Expression Graph (PEG)*** *is a directed graph* $(\mathcal{N}, \mathcal{E}, label, n_0)$ *with* $\mathcal{N}$ *a set of nodes,* $\mathcal{E}$ *a set of edges between nodes,* $label : \mathcal{N} \to \mathcal{F}$ *a labeling function from nodes to symbol functions and* $n_0 \in \mathcal{N}$ *is a "root" node. A PEG satisfies the following assertions:*

- *the edges* $e_1, ..., e_i$ *stemming from a node* n *are ordered;*
- *a node* n *is the source of* i *edges if and only if its function symbol* $label(n)$ *is of arity* i*.*

In essence, a PEG represents a term t, given by root node n_0. Each node n is a subterm $f(t_1, ..., t_i)$ of t, characterized by its function symbol f of arity i and its direct subterms $t_1, ..., t_i$ represented by the

nodes targeted by the edges $e_1, ..., e_i$ stemming from n. It is common to consider PEGs with **sharing**: if two nodes represent the same term, then they are the same node. This design choice allows efficient implementations of PEGs.

The PEG associated to the term of Figure 18 *(a)* is shown in *(b)*. The root node is represented on the top. The four occurrences of the subterm 1 in the term share the same node in the PEG.

For our ends, we want to represent not only terms, but equivalence classes of terms. The EPEG is an efficient data structure for representing those equivalence classes:

Definition 2 (Equality-PEG)**.** *An **Equality-PEG (EPEG)** is a structure $(\mathcal{N}, label, \mathcal{C}, \mathcal{E})$ with $\mathcal{N}$ a set of nodes (now called **e-nodes**), $label : \mathcal{N} \to \mathcal{F}$ a labeling function from e-nodes to symbol functions, $\mathcal{C}$ a partition of $\mathcal{N}$ into equivalence classes (called **e-classes**) and $\mathcal{E}$ a set of edges from e-nodes to e-classes. An E-PEG satisfies the following assertions:*

- *the edges $e_1, ..., e_i$ stemming from an e-node n are ordered;*
- *an e-node n is the source of i edges if and only if $label(n)$ is of arity i;*
- *if two e-nodes n and m have the same label $label(n)$ and the same children e-classes, then n and m are the same e-node.*

EPEGs can be seen as a mix of some PEGs and an union-find structure. In contrast with PEGs, in an EPEG edges point towards e-classes rather than e-nodes. An e-class represents the variety of equivalent terms represented by its e-nodes. An e-node n represents terms constructed with its operator $label(n)$ and by picking some terms represented by its children e-classes for the direct subterms. Intuitively, it's as if we were recursively constructing a term and were given several equivalent options at each depth level.

The two first assertions ensure that the e-nodes behave like the function symbol they are labeled with. The last assertion of the definition corresponds to the sharing we evoked for PEGs. In an EPEG, sharing is very natural due to a principle we call **congruence**: if t_1 and t_2 are two terms that have the same function symbol and which immediate subterms are pairwise equivalent, then t_1 and t_2 are equivalent. This means the nodes n_1 and n_2 representing t_1 and t_2 would have to be in the same e-class: at this point this e-class contains two e-nodes that have the exact same label and children, so we might as well fuse them.

Figure 18 *(c)* shows an EPEG which represents only the term in *(a)*. It's really similar to the PEG of *(b)* but edges now point towards the e-classes, which we represent with dotted line boxes. The EPEG in *(d)* represents slightly more terms because we enriched e-classes X and Y with new equivalent e-nodes. The set of all four equivalent terms represented by the top Φ-node is in *(e)*: note that it corresponds to all possible ways to pick a combination of equivalent terms in the e-classes that have more than one e-node. More on what criterion exactly makes two terms equivalent in the next subsections.

A PEG is essentially an EPEG in which each e-node has its own e-class and an e-node is singled out as the root node: for the sake of uniformity, in this document we use the EPEG representation with dotted lines e-classes even for PEGs and always represent the root node on the top.

7.2 Semantics of EPEGs

To define the semantics of EPEGs, we need to define the semantics of the terms over $\mathcal{F}$ they represent.

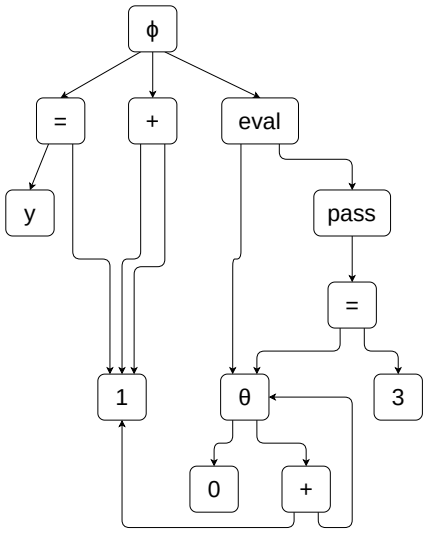
The general intuition behind the semantics of our terms is that each term represents a computation. This computation yields a value which can be a domain value, for example integer or boolean or a special value $\perp$ that means "failure to compute" (we say that *the values are $\perp$-lifted*). Some of these computations' values might also depend on which loop iteration we are at: the semantics of a term is the sequence of values it takes at each iteration of the loop (we say that *the values are also loop-lifted*).

Formally, for t a term and i an integer, we denote $[\![t]\!](i)$ the value of term t at loop iteration i. This value can be computed according to the compositional semantics of Figure 19 which we give a breakdown of:

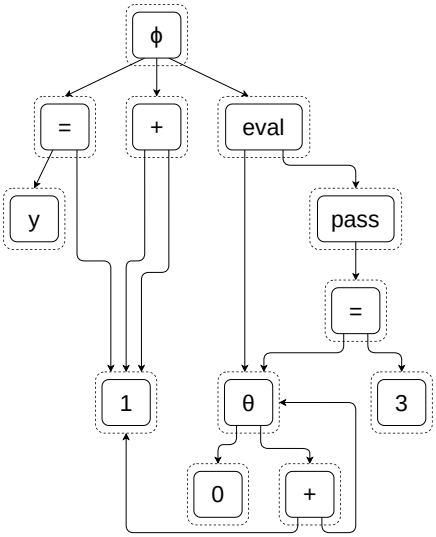
$$\Phi(equal(y,1), plus(1,1), eval(t, pass(equal(t,6))))$$

where $t = \theta(0, plus(1,t))$

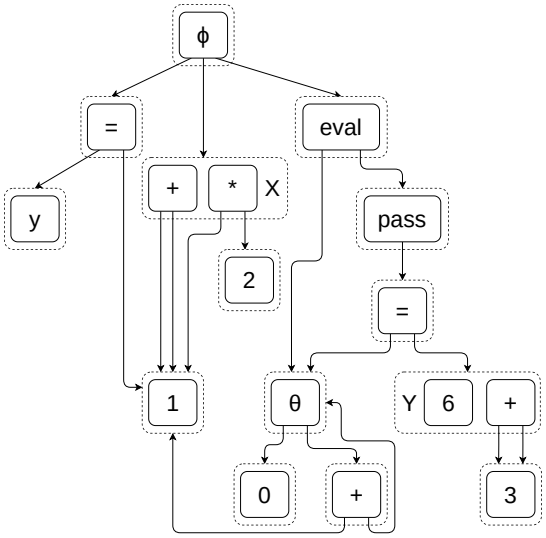
(a)



(b)



(c)



(d)

$$\{t_1 \equiv t_2 \equiv t_3 \equiv t_4\} \text{ where}$$

$$\begin{aligned}
t_1 &= \Phi(equal(y,1), plus(1,1), eval(t, pass(equal(t,6)))) \\
t_2 &= \Phi(equal(y,1), plus(1,1), eval(t, pass(equal(t, plus(3,3))))) \\
t_3 &= \Phi(equal(y,1), mult(1,2), eval(t, pass(equal(t,6)))) \\
t_4 &= \Phi(equal(y,1), mult(1,2), eval(t, pass(equal(t, plus(3,3))))) \\
t &= \theta(0, plus(1,t))
\end{aligned}$$

(e)

```
1  foo(y) {
2      x = 0
3      while (x !=6) {
4          x = x + 1
5      }
6      if (y = 1) {
7          return 2
8      } else {
9          return x
10     }
11 }
```

(f)

Figure 18: Examples of terms, PEGs, EPEGs and associated program.

$$[\![plus(t_1,t_2)]\!](i) = \begin{cases} \bot \text{ if } [\![t_1]\!](i) = \bot \text{ or } [\![t_2]\!](i) = \bot \\ [\![t_1]\!](i) + [\![t_2]\!](i) \text{ otherwise} \end{cases} \qquad [\![or(t_1,t_2)]\!](i) = \begin{cases} true \text{ if } [\![t_1]\!](i) = true \\ \bot \text{ if } [\![t_1]\!](i) = \bot \\ [\![t_2]\!](i) \text{ otherwise} \end{cases}$$

$$[\![\Phi(c,t_1,t_2)]\!](i) = \begin{cases} \bot \text{ if } [\![c]\!](i) = \bot \\ [\![t_1]\!](i) \text{ if } [\![c]\!](i) = true \\ [\![t_2]\!](i) \text{ if } [\![c]\!](i) = false \end{cases}$$

$$[\![\theta(t_1,t_2)]\!](i) = \begin{cases} [\![t_1]\!](i) \text{ if } i = 0 \\ [\![t_2]\!](i-1) \text{ otherwise} \end{cases} \qquad [\![eval(t_1,t_2)]\!](i) = \begin{cases} \bot \text{ if } [\![t_2]\!](i) = \bot \\ [\![t_1]\!]([\![t_2]\!](i)) \text{ otherwise} \end{cases}$$

$$[\![pass(c)]\!](i) = \begin{cases} \bot \text{ if } E = \emptyset \\ min(E) \text{ otherwise} \end{cases} \quad \text{where } E = \{j \in \mathbb{N} | [\![c]\!](j) = true \ \wedge \ \forall k < j, [\![c]\!](k) \neq \bot\}$$

Figure 19: Compositional semantics for PEG operators.

- **domain operators**: we included the computation rules for two example common operators. The formula for *plus* is the $\bot$-lifted ($\bot$ are propagated) and loop-lifted (computation is done at a given loop iteration) standard compositional semantics of *plus*. We do not show the formula for all operators, but most of them are similarly $\bot$-lifted and loop-lifted. The *or* is slightly more complex: the evaluation of subterms is lazy and does not propagate a $\bot$ from the right subterm if the left subterm is evaluated as true.
- **the Φ operator**: Φ is the conditional operator. Its first subterm acts as a condition and is evaluated. If it is evaluated as true, then we evaluate the second subterm (the "**then-branch**" of the Φ), if it is false we evaluate the third subterm (the "**else-branch**" of the Φ).
- **the θ loop operator**: θ is the operator that allows to build terms with sequences of values. If we are at the first loop iteration then the θ-term has the same value as its first subterm. Otherwise, we evaluate the second subterm **at the previous loop iteration**. This allows to recursively build a sequence of values, notably if the second subterm refers to the θ-term.
- **the *eval* loop operator**: *eval* is the operator computing the value of a given term at a specific loop iteration. The second subterm is evaluated to obtain the loop iteration we are interested in. The first subterm will be evaluated at this given loop iteration.
- **the *pass* loop operator**: *pass* is the operator that computes the first loop iteration at which a condition is true. It takes a boolean subterm and its value is the first loop iteration at which the boolean subterm is evaluated as true, if it exists, $\bot$ otherwise. An additional subtlety: if the subterm has value $\bot$ at a loop iteration before becoming true, then the *pass*-term is evaluated as $\bot$ too. This mimics the behaviours of a while-loop: suppose a loop iteration does not compute, then the loop itself does not compute and the exit condition is never actually met. Indeed, *eval* and *pass* are commonly used in tandem to represent such loops.

The top level of a computation is not inside a loop so it does not matter at which loop iteration we evaluate the root of the term: it is standard to pick loop iteration 0. Let us for example study the computation represented by the term in Figure 18 *(a)*. The root is a Φ-node, hence a conditional. If the

input of the program is equal to 1, then our term has value $1 + 1 = 2$. Otherwise we evaluate the *eval* subterm. The θ term named t corresponds to a sequence of value 0 at loop iteration 0 and that increments its previous value at each successive iteration. We evaluate this sequence at the first loop iteration for which it has value 6, *i.e.* at loop iteration 6, and t has value 6. We represent in *(f)* a program that would correspond to the term.

If the EPEG manipulated correctly, the terms represented in the same e-class are equivalent, *i.e.* have the same value. The concept of semantic value can be generalized to the e-nodes and e-classes.

7.3 Algorithmics of EPEGs - patterns and ematching

EPEGs support a variety of operations that allow querying and modification of the set of terms. The EPEG data structure has the following interface:

- **finding** if a term is represented by the EPEG, and the e-class that represents it if so;
- **merging** two e-classes into one (all the terms they represent are now equivalent);
- **adding** a new e-node to the EPEG, in its own new e-class.

A fourth main operation exists, called **ematching**. Ematching is about searching for terms that fit a certain pattern in the EPEG.

Definition 3 (Pattern)**.** *A **pattern** P is a term built over the function symbols of the term language F and some variable placeholders from a set of variables $\mathcal{V}$:*

$$P := x \in \mathcal{V} \mid f(P_1, ..., P_i) \text{ for } f \in \mathcal{F} \text{ of arity } i$$

Essentially, a pattern describes a general structure of term. For example we could look for the pattern $plus(x, x)$ (where x is a variable placeholder). This pattern matches with any term that is the *plus* of two identical subterms. In some cases, we will be interested in specific kinds of patterns such as **boolean patterns**: patterns that either have a logic operator at the root or are just a variable placeholder.

Several possible e-classes in an EPEG may contain terms that fits the pattern P. The concept of occurrence characterize such fits:

Definition 4 (Occurrence)**.** *An **occurrence** of a pattern P is a couple (X, σ) with X an e-class and σ a substitution of the variable placeholders of P by e-classes such that $\sigma(P)$ is indeed represented in e-class X.*

The variable placeholders of P must be replaced by e-classes to obtain a real term that we could fit with a part of the EPEG. To that end we use a substitution σ from variable placeholders to e-classes. For instance, if we were to search pattern $plus(x, x)$ in the EPEG of Figure 18 *(d)* there would be two occurrences: one in class X with the class of 1 as x and one in class Y with the class of 3 as x.

All these operations allow to define equivalence rules: some pattern of terms are equivalent to others, and the EPEG can be enriched with these new equivalences.

Definition 5 (Equivalence rule)**.** *A **rewriting rule** is a couple (P, P') with P a pattern and P' another pattern over the variables of P. We often write $(P \sim P')$.*

A rule (P, P') states equivalence between pattern P and pattern P': if the rule is sound, then for any substitution σ of the variables of P, $\sigma(P)$ should be equivalent to $\sigma(P')$. For instance, we could create the rule $(plus(y, y) \sim mult(y, 2))$: adding anything with itself is the same as multiplying it by 2.

Peggy is based on the equality saturation approach, that aims to enrich an EPEG by successively applying such equivalence rules. No more background should be necessary to understand the next section in which we improve the expressivity of Peggy for our Java-Kotlin translation problem.

8 Case study of the loop-for program equivalence

With its base set of equivalence rules, Peggy is unable to prove equivalence between the two control flows of Figure 16. In this section we study the PEGs associated to these programs and build the equivalence rules needed to prove equivalence.

8.1 Equivalence check for the atomic loop-for program

First we consider an atomic program presenting this loop-for structure. Figure 20 shows a program in which the body loop is a variable increment, in Java and Kotlin, and the associated bytecodes. They have different but equivalent control flows.

Let us take a look at the PEGs associated with these bytecodes in Figure 20.

- In the Java program, variables i and z have the same value at each iteration of the *while*-loop. In the associated PEG, the θ-node represents both these sequences of values. The program is an evaluation of the value of z at the first loop iteration for which the value of z is greater than x.
- In the Kotlin program, z is actually $i+1$ at each iteration of the *while*-loop. In the PEG, the θ-node represents the sequence of values of i and the +-node represents the sequence of values of z. The *eval*-node computes the value of z at the first loop iteration for which the value of i is equal to x. This corresponds to the structure inside the conditional in the Kotlin bytecode of Figure 20. The program is represented by the Φ-node: compute the *eval* if the initial value of i is not above x, otherwise return 1 (the initial value of z).

Our goal is now to show that both PEGs are equivalent. We must design one (or several) equivalence rules that will allow Peggy to merge the root e-classes during equality saturation. There are two important criteria to keep in mind when designing a rule: it must be *sound* of course (the two sides of the rule must actually be equivalent), and it should be somewhat *generic* (we could make a rule that essentially says "those exact two terms are equivalent" but that would not be very useful for other programs). Intuitively, a rule is generic if it represents one single point of reasoning, stated as generally as possible.

An important point of reasoning to prove that our programs are equivalent is that variable i takes successively each possible integer value greater than its initial value 1, since it is incremented every loop iteration. The first iteration at which i goes above value x is therefore:

- the 0th iteration if i starts above x (*i.e.* $1 > x$);
- the iteration just after the first (and only) iteration at which i reaches value x otherwise.

In general, this is true for any sequence of integer values that is incremented at each iteration: we design the rule of Figure 21 to represent this exact point of reasoning. Terms i_0 and i_f represent integers, but are otherwise universally quantified. The leftmost side of the equivalence is "the first iteration at which a sequence goes over i_f". On the rightmost side, the *pass*-term is "the first iteration at which said sequence reaches i_f". The Φ-term represents the case study of whether $i_0 > x$: if so it return 0, otherwise return the successor of the iteration described by the *pass*-term.

Peggy can now prove equivalence between the two programs, using our new rule and three more rules that state how the PEG function symbols interact. We represent this proof in Figure 22: for the sake of clarity, we only show a succession of PEGs, each obtained by applying a rewriting rule from the previous one, rather than an EPEG. Step by step we have:

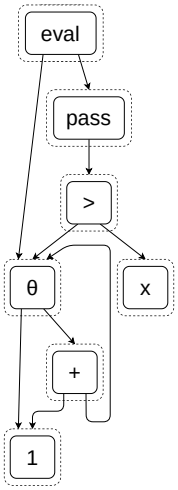
- from the Java PEG in Figure 20 to *(a)*, apply the rule of Figure 21. This replaces the *pass*-node by the Φ-structure of the rightmost side of the rule;
- from *(a)* to *(b)*, swap the *eval*-node and the Φ-node using a distribution rule. The root of the PEG is now a Φ-node and there is a different *eval* on each of the branches of the Φ;

```java
//Java program
int main(int x) {
    int z = 1;
    for (int i = 1; i <= x; i++) {
        z++;
    }
    return z;
}
```

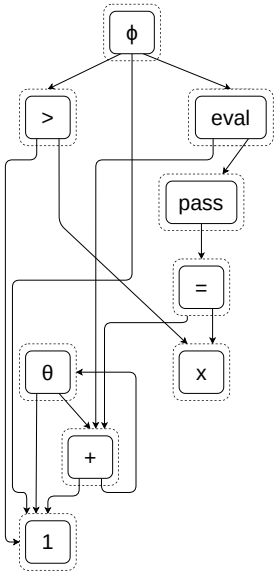
```
#Java bytecode
z = 1
i = 1
while (i <= x) {
    increment z
    increment i
}
return z
```

```kotlin
//Kotlin program
fun main(x: Int): Int {
    var z = 1
    for (i in 1 until x) {
        z++
    }
    return z
}
```

```
#Kotlin bytecode
z = 1
i = 1
if (i <= x) {
    increment z
    while (i != x){
        increment i
        increment z
    }
}
return z
```



(a) Java PEG.



(b) Kotlin PEG.

Figure 20: Atomic Java and Kotlin programs presenting the problematic loop-for structure, and pseudo-code for associated bytecodes.

$$\forall i_0, i_f \text{ representing integers,}$$

$$(\ pass(gt(i, i_f)) \ \sim \ \Phi(gt(i_0, i_f), 0, succ(pass(equal(i, i_f)))) \ )$$

$$\text{where } i = \theta(i_0, plus(1, i))$$

Figure 21: A new equivalence rule to solve the loop-for problem.

- from *(b)* to *(c)*, resolve the *eval* of the then-branch of the *Phi*: evaluating a sequence at the 0th iteration returns the head of the sequence, *i.e.* 1 in this case;
- from *(c)* to *(d)*, simplify the *eval* of the else-branch of the Φ: evaluating a sequence at the $n+1$th iteration is the same as evaluating its tail at the nth iteration. In this case, the *eval*-node now points toward the tail of the θ-node: the +-node. We reached the Kotlin PEG.

The new equivalence rule allows Peggy to prove equivalence for most simple programs containing a problematic for-loop structure. In the next subsection we review an example of a program in which the new equivalence rule is not sufficient.

8.2 Dealing with more complicated programs

Any program in which the for-loop body contains a break condition has a PEG structure more complex than the PEG of the previous subsection. Figure 23 shows an example of such complex PEG. At this point, PEGs start to be somewhat large so we only present fragments of them, or their general structure. Here we write in e-classes boxes the computation they represent. Our program is a for-loop with an exit condition $i > z$ and an internal break condition $x = 5$.

It is easier to understand the associated control flow graph on the Java PEG, in *(b)*: the loop body is evaluated at the first iteration that either satisfies the break condition, or the standard loop exit condition (hence the *or* boolean structure). The value of the loop body depends on which condition was satisfied, hence the check with a Φ-node.

The Kotlin PEG in *(c)* has the special Kotlin loop-for compilation structure. The break condition impacts this structure in similar ways as for the Java PEG: the *pass*-node points to an *or*-structure of the breack condition and the loop exit condition (an equality $i = z$ in Kotlin); the body of the loop is a Φ-node that checks which condition was satisfied.

Peggy cannot prove equivalence, even with our new equivalence rule from Figure 21. There are two main problems that need to be solved in this case:

- the rule cannot be applied immediately: the *or*-node is between the *pass* and the condition $i > z$;
- fixing the problem on the *pass* side is not enough, both loop bodies are different: the Φ-nodes that check which condition led to the end of the loop use a different check to do so ($i > z$ and $x \neq 5$ respectively).

We spent some time during the internship investigating this PEG pair and trying to prove their equivalence. In this exploratory process, we try to make the two PEGs equal: we would repeatedly pick a PEG and replace a part with an equivalent one, by using an existing equivalence rule or by designing a new one. We succeeded in rewriting the two PEGs into equality. Some of the rules we used can be found in Figure 49 (in Annex): some of them were already present in Peggy. The proof is very long, even with human oversight, so we only describe the most relevant parts.

Dealing with the *or*: the overall proof sketch is a case disjunction over which condition triggers before the other and ends the loop. This can be achieved by breaking the *or* using a new equivalence rule of Figure 24: the first iteration at which $C \vee D$ is true can be obtained by checking which of C and D becomes true first (with a Φ), and then returning the minimum of the two.

The main use of this rule is to replace the *or* by a Φ operator, which can then be swapped with other operators with structural equivalence rules. This allows to perform the case disjunction on other parts of the PEG. The rule also reunites the *pass*-node with the condition $i>z$, which allows to use our previous equivalence rule for the for-loop.

Performing the case disjunction: once the Φ-node of case disjunction is set up, we can exploit the knowledge gained in each case. We performed many transformations of the PEG to get both of them to have the same loop body.

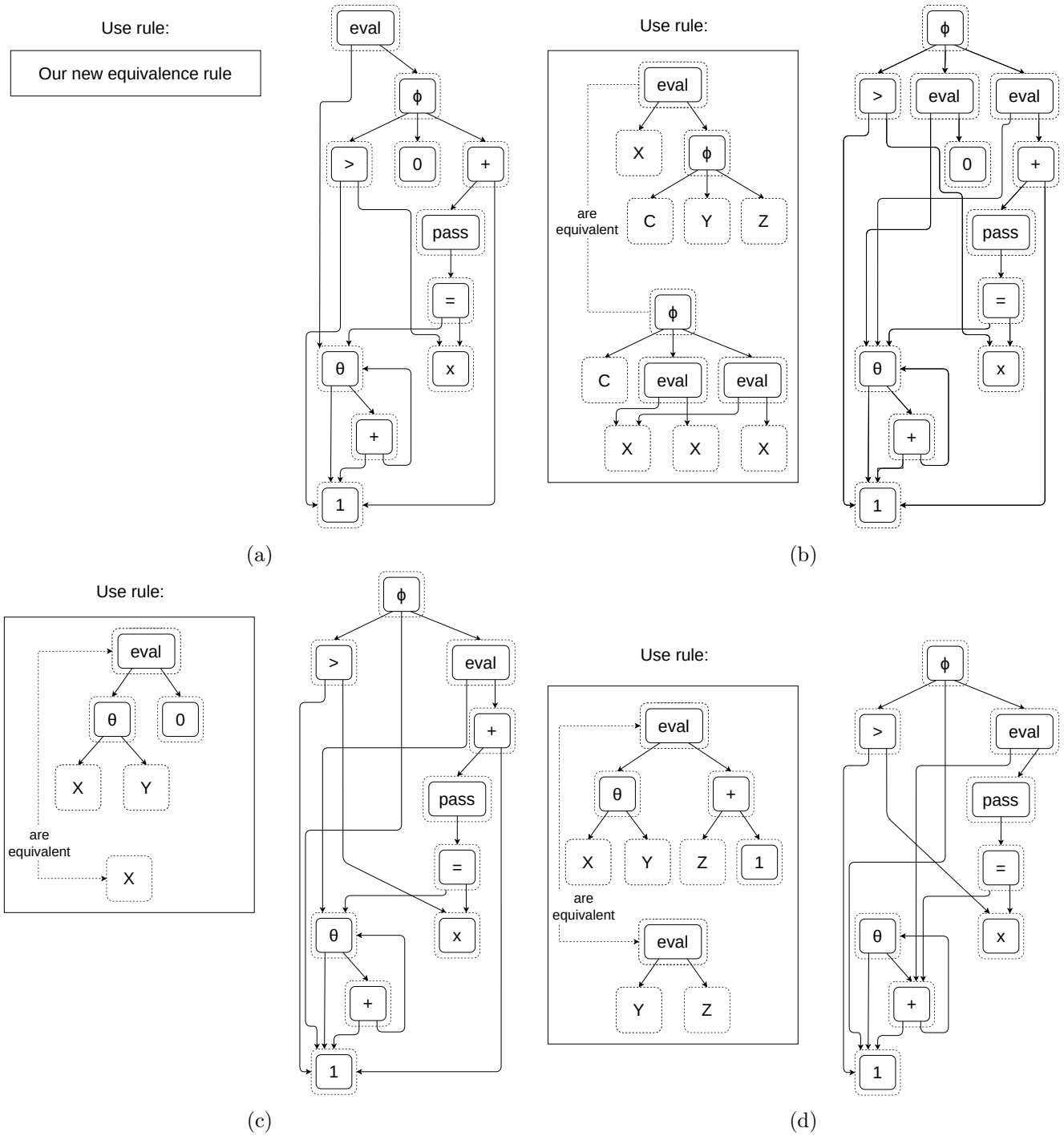
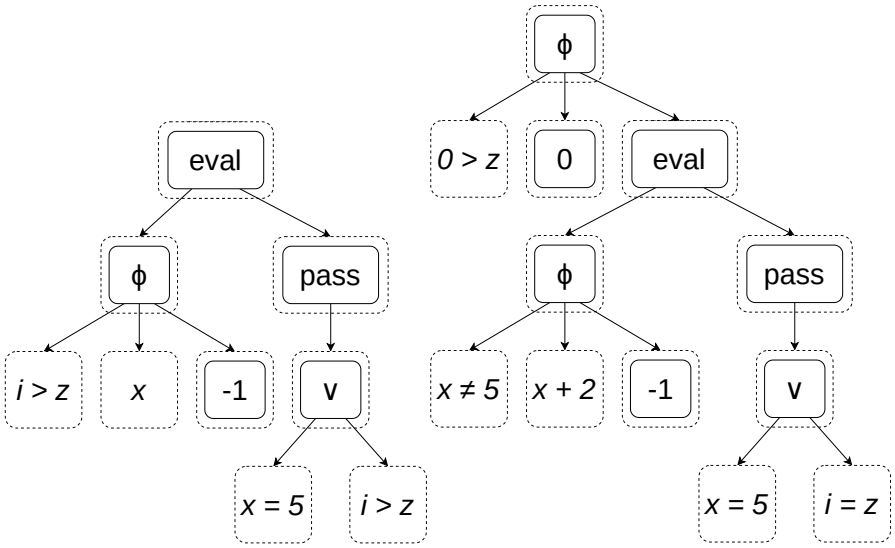


Figure 22: Four-step rewriting of the Java PEG into the Kotlin PEG.

```
1  foo(z) {
2      x = 0
3      for (i=0; i<=z; i++) {
4          if (x = 5) {
5              return -1
6          }
7          x = x+2
8      }
9      return x
10 }
```

(a)



(b) Java PEG. (c) Kotlin PEG.

Figure 23: A program with a break condition inside the loop, and general structure of the associated PEGs.

$$\forall c, d,\ (\ pass(or(c,d))\ \sim\ \Phi(gt(pass(c), pass(d)), pass(d), pass(c)\ )$$

Figure 24: New equivalence rule for breaking a *pass-or* structure.

One such transformation is shown on Figure 25 in a new equivalence rule. If we know that c is true for the first time strictly after iteration x, we can deduce that c is false at iteration x. Hence, when evaluating c at iteration x on the then-branch of a Φ with this condition, we can replace c with *false*.

We could not get Peggy to perform our equivalence proof even when empowered with all our new equivalence rules: the engine reached timeout. A weakness of a solver-like approach such as Peggy is that it can be hard to pinpoint exactly why the engine could not prove the equivalence. A likely cause is that the proof is very long in this case: this could be a performance issue. In the next subsection, we discuss a possible way to improve these performances.

8.3 Limit of the approach

One thing we noticed when doing a by hand equivalence proof on PEGs is that we used a lot of "swapping" structural rules. Many times, a relevant transformation of the PEG requires a very specific pattern that has to be created by moving the function symbols around. Most of the time, we even swapped back to recreate the initial structure once the transformation was performed.

A typical transformation that is tedious to to actually perform is the one of Figure 25. This pattern almost never occurs naturally, so we must swap operators around to get the $eval(c, x)$ to the root of such a Φ-node. This creates an important overhead in the proof and leads to a loss of performance.

$$\forall c, w, z,\ (\ \Phi(gt(pass(c), w), eval(c, w), z)\ \sim\ \Phi(gt(pass(c), w), eval(false, w), z)\ )$$

Figure 25: A new rule that exploits knowledge about which condition triggers first.

This observation gave us an idea of an algorithmic improvement for equality saturation. When reasoning directly about a program, it is natural to exploit the information gained in the condition of the *if* structures. Granted some immutability of x, it is possible to perform constant propagation on all instances of x inside a *if* $(x = 5)$ structure for instance.

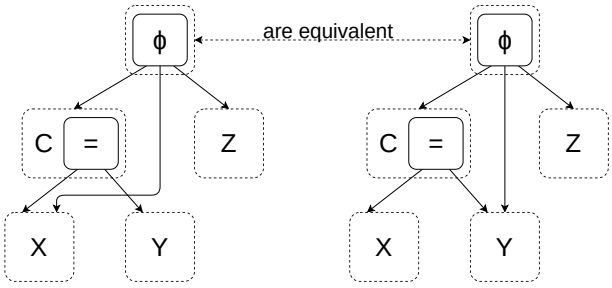
In contrast, our equivalence rules are very local, and require the terms they perform a transformation on to be very close in the PEG: we would need an instance of x to be right under the Φ-node corresponding to *if* $(x = 5)$ to change x to 5. If we could somehow propagate information (*i.e.* **context**) from the terms that generate it (for example the Φ) to the terms that benefit from it, we could potentially avoid most of the operator swapping.

Once we had this idea, we decided to focus the internship on algorithmic improvement of equality saturation. The next section presents the technical heart of our algorithmic framework for contextual equivalence rules support.

9 Contextual pattern/telepattern support for EPEGs

We want our equivalence rules to be able to exploit contextual information, for example the information gained in Φ-nodes. The most basic example of such a rule is given in Figure 26: the Φ-node checks if X and Y are equals and this information can be used as a context in the *then*-child.

```
1  if (X=Y) {
2      X
3  } else {
4      Z
5  }
```

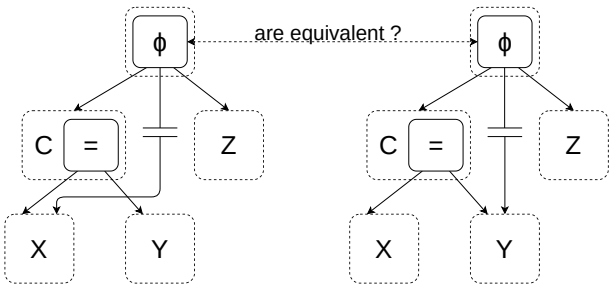



```
1  if (X=Y) {
2      Y
3  } else {
4      Z
5  }
```

Figure 26: Two equivalent EPEG terms and their associated programs. We use the underline notation in programs to represent the computation associated to a class from the EPEG.

However, as written, the rule of Figure 26 can only be used if class X is the immediate *then*-child of the Φ-node. In practice this will rarely be the case in an EPEG obtained from a real program: X will often appear as a subcomputation in a bigger term, not right under the Φ-node. We are closer to the situation of Figure 27 where the instance of X is further along the *then*-branch in the PEG.

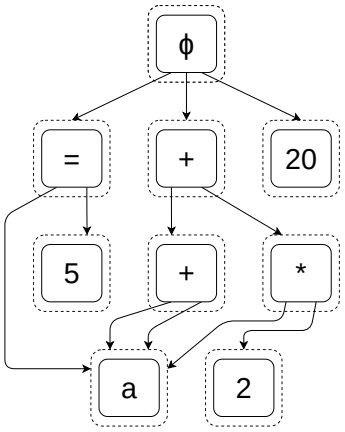
```
1  if (X=Y) {
2      ... X ...
3  } else {
4      Z
5  }
```




```
1  if (X=Y) {
2      ... Y ...
3  } else {
4      Z
5  }
```

Figure 27: A variation of the programs and EPEGs of Figure 26 where the instance of X (resp. Y) is a subcomputation in the *then*-branch of the Φ-node. The crossed edge notation denotes the same term structure in both EPEGs, except for the instance of X replaced by an instance of Y.

The transition from the rule of Figure 26 to the eventual rule of Figure 27 is not immediate for two main reasons.




```
1 foo (a) {
2     if (a = 5) {
3         return (a+a)+(a*2)
4     } else {
5         return 20
6     }
7 }
```

Figure 28: Our main example EPEG.

- The equivalence between the two programs of Figure 27 is not guaranteed and depends on the actual structure of the branch that links the Φ-node to X: C must hold still when the computation uses X.
- If the programs indeed are equivalent, then the two equivalent classes **are not** X **and** Y but rather the classes of the Φ-nodes (X and Y are equivalent but only **under the condition that** C **holds**). The rule must therefore encompass the Φ not just X and Y. The pattern of the rule of Figure 27 cannot be explicitly specified using the standard pattern language since it must allow branches from Φ to X that are of arbitrary length. We must also specify that the branch from Φ to Y on the right side of the equivalence is "the same" as the then-branch from Φ to X on the left side.

We call those rules involving an arbitrary long branch **contextual rules** as they rely on some boolean condition being true in the context in which a class is computed. Using the notion of contexts and a variety of algorithms, it is possible to determine when such a rule can be applied, and then proceed with the rewrite.

In this section a **second simplification** is in action, we only present the case of an EPEG without any loop operator (θ, *eval*, *pass*, ...) for the sake of pedagogy. We can generalize the framework to all EPEGs, and we give insight on how to do so in Section 10.1. For the time being, we only consider conditionals, arithmetic and logic operations and the program input variables. A typical example of such a program is presented in Figure 28: through this section we will regularly showcase our concepts and algorithms on this EPEG.

9.1 Possible contexts and contextual equivalence rules

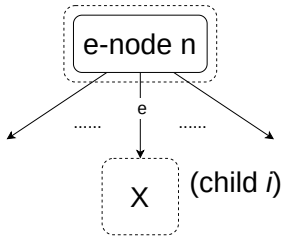
The key concept that empowers the contextual equivalence rules is the concept of **possible context** of an e-class. For loopless programs, this concept is strongly tied to the then-branch of Φ-nodes and paths. We use the notations presented in Figure 29 in order to describe the characteristics of the edges constituting a path. In an EPEG, **a path from an e-node** n **to an e-class** X is a finite series of edges $(e_0, e_1, ..., e_i)$ such that:

- $source(e_0) = n$ (the path begins in n);
- $target(e_i) = X$ (the path ends in X);
- for all $j < i$, $source(e_{j+1}) \in target(e_j)$ (each edge of the path leads to the e-class in which the next one has its source).

$source(e)$ ($\in \mathcal{N}$) is the source e-node of e

$target(e)$ ($\in \mathcal{C}$) is the target e-class of e

$number(e)$ ($\in \mathbb{N}$) is the index of e among the edges leaving $source$



$source(e) = n$

$target(e) = X$

$number(e) = i$

Figure 29: Base notations to fully describe an edge in an EPEG.

This notion can be generalized for paths starting in an e-class Y (in which case we require $source(e_0) \in Y$) and for paths leading to an e-node m (in which case we require $m \in target(e_i)$).

We want to give a special status to the edges that define the then-branches of Φ-nodes: for every edge e, we define a new notation $cond(e)$ (as in: **"the condition to take** e**"**) such that

$$cond(e) = \begin{cases} C \text{ if } source(e) = \Phi(C, X, Y) \text{ and } number(e) = 2 \\ \cdot \text{ otherwise} \end{cases} \qquad \in \mathcal{B} \cup \{\cdot\}$$

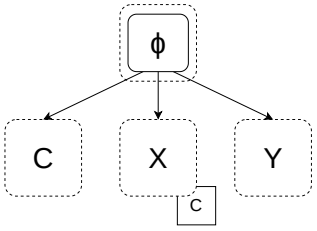
Essentially $cond(e)$ is $\cdot$ (a special value that means "no condition") for all edges except for the second edges leaving Φ-nodes, in which case $cond(e)$ is the condition of the Φ-node. We also generalize the $cond$ notation to paths. $cond(\pi)$ is the set of all conditions of the edges of π:

$$cond((e_0, e_1, ..., e_i)) = \{cond(e_j) | j \leq i \ \wedge \ cond(e_j) \neq \cdot\} \in 2^{\mathcal{B}}$$

Definition 6 (Possible context)**.** *Let χ be a set of boolean e-classes, n an e-node and X an e-class. We say that* ***from*** *n,* ***χ is a possible context for*** *X in a loopless EPEG $\mathcal{P}$, if there exists in $\mathcal{P}$ a path $\pi = (e_0, e_1, ..., e_i)$ from n to X such that $\chi \subseteq cond(\pi)$. Such π is named a* ***context-bearing path*** *for χ.*

This definition can be intuitively understood this way: from n, χ is a possible context for X in the EPEG $\mathcal{P}$ if and only if $\mathcal{P}$ represents at least a computation (the one of e-node n) in which the computation of X is used (at the end of the path π for example) under the scope of all conditions of χ being true (since it is in the then-branch of a Φ-node for each condition of χ).

One of the simplest example of a possible context is the then-branch of a conditional. Figure 30 represent a basic Φ-node structure in which the Φ-node is a source of possible context $\{C\}$ for class X: the path exhibiting this property is simply the second edge leaving the Φ-node.



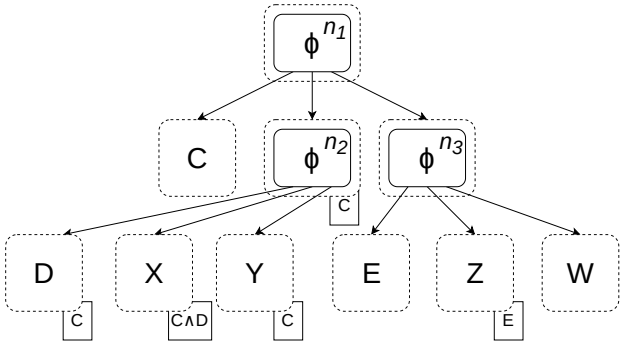

```
1  if (C) {
2      X
3  } else {
4      Y
5  }
```

Figure 30: A simple *if-then* program and the associated EPEG annotated with a possible context (in the square box). We denote $\{C\}$ as C for the sake of simplicity.

Figure 31 shows an example of a non-singleton possible context: $\{C, D\}$ is a possible context for class X from n_1. Since both C and D are known to be true for this use of X, we often use the notation $C \wedge D$ to describe context $\{C, D\}$. Although we did not indicate it on the figure, note that $\{C\}$ is also a possible context for class X from n_1 and the same can be said for $\{D\}$. In general the following lemma holds:

Lemma 1. *Let $\chi, \chi' \in 2^{\mathcal{B}}$ be two contexts such that $\chi' \subseteq \chi$. If χ is a possible context for X from n then χ' is also a possible context for X from n.*

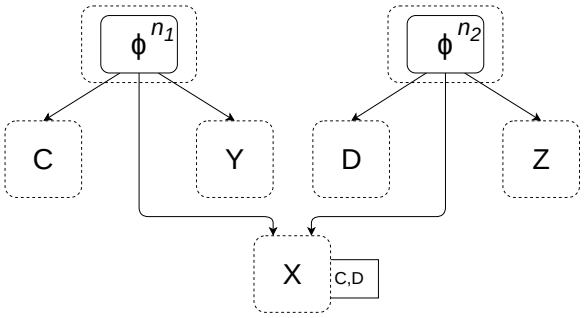
Proof. Let π be a path from n to X such that $\chi \subseteq cond(P)$. Since $\chi' \subseteq \chi$, $\chi' \subseteq cond(\pi)$ too. □




```
1 if (C) {
2      if (D) {X}
3      else {Y}
4 } else {
5      if (E) {Z}
6      else {W}
7 }
```

Figure 31: Nested-*if* EPEG annotated with some possible contexts. The three Φ-nodes are given an identifier for better clarity. On figures we will mostly use the $\wedge$ notation: $C \wedge D$ stands for $\{C, D\}$.

It is also important to note that an e-class X can have possible contexts stemming from different, non-simultaneous uses of the computation of X. There is a natural sharing in the EPEG which means that all uses of X in the overall computation refer to the same e-class X; not all of these uses are going to be with the same contexts: this is why we call the concept *possible* context. Figure 32 presents a simple case of two possible non-simultaneous contexts: class X has possible context $\{C\}$ from n_1 and possible context $\{D\}$ from n_2.




```
1 if (C) {
2      if (D) {X}
3      else {Y}
4 } else {
5      if (E) {Z}
6      else {W}
7 }
```

Figure 32: An EPEG representing several uses of X within different contexts. X has possible context $\{C\}$ and possible context $\{D\}$ but not possible context $\{C, D\}$.

The relation between n and X expressed by the affirmation "from n, χ is a possible context of X" is a relation of ancestry: n is a special kind of ancestor of X. Many other e-nodes m will share this property with n. In fact the following lemma holds:

Lemma 2. *Let n and m be two e-nodes, with m a parent of the e-class of n. If χ is a possible context for X from n, then χ is a possible context for X from m too.*

Proof. Consider a path from m to n and extend it with a path π from n to X that verifies $\chi \subseteq cond(\pi)$. The obtained path π' trivially verifies $\chi \subseteq cond(\pi')$. □

Due to Lemma 2, the concept of possible contexts encompasses many irrelevant relationships between e-nodes and e-classes. We wish to characterize the "closest to X" n from which χ is a possible context, some sort of frontier of e-nodes from which the context stems.

Definition 7 (Source of a possible context)**.** *We say that n* ***is a source of possible context*** *χ* ***for*** *X if there exists a path $\pi = (e_0, e_1, ..., e_i)$ from n to X such that χ is* ***not*** *a possible context for X from any e-node of $target(e_0)$, and $\chi \subseteq cond(\pi)$.*

Some examples of source and non-source e-nodes for possible context can be found in Figure 31. From n_1, $\{E\}$ is a possible context for Z. The same can be said from n_3. Since the only path from n_1 to Z goes through n_3, n_1 is not a source of possible context $\{E\}$, but n_3 is.

The concept of possible context can be applied to implement the contextual rules. If each e-class is annotated with some of its possible contexts, the pattern of Figure 27 can be detected for arbitrary length of paths between the Φ-node and X. We can now define what is a **contextual pattern**, the central component of a contextual equivalence rule:

Definition 8 (Contextual pattern)**.** *A **contextual pattern** is a couple (P, ρ) with P a standard pattern (the **structural requirement** of the pattern), and ρ a set of boolean patterns (the **contextual requirements** of the pattern).*

Intuitively, P is the term-structure we are looking for and ρ is a set of conditions we must find as a possible context of the term. For example, the contextual pattern implemented in the leftmost part of the rule of Figure 27 would be:

$$(x, \{equal(x, y)\}) \text{ with } x, y \text{ two variable placeholders}$$

as in "any term, under the context of this term being known to be equal to some other term". For the sake of brevity and mainly in code, we may sometimes refer to contextual patterns as **telepatterns** (in reference to the arbitrary length of the branch between the structural requirement and the source of the possible context). In order to give meaning to this definition, we also need to define what constitutes an occurrence of a contextual pattern:

Definition 9 (Occurrence of a contextual pattern)**.** ***An occurrence of contextual pattern** (P, ρ) is a couple (X, χ, σ) with X an e-class, χ a context and σ a substitution over the variables of P and of ρ such that:*

- *e-class X indeed represents the term $\sigma(P)$;*
- *each boolean term of $\sigma(\rho)$ is indeed represented in χ;*
- *X has possible context χ.*

This definition is somewhat similar to the occurrences of a standard pattern but instead of just exhibiting a X that contains $\sigma(P)$, we do the same for the context requirements and also check that exhibited context χ is indeed a possible context of X. Figure 33 presents a simple instance of occurrence of a contextual pattern $(x, \{equal(x, y)\})$. There is only one occurrence in class X, contextual requirements matched in $\{C\}$ with substitution $x \leftarrow X, y \leftarrow Y$: the term a exists in an e-class which has context $\{equal(a, 5)\}$.

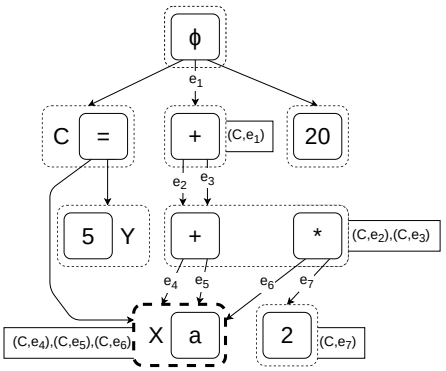


Figure 33: Occurrence of contextual pattern $(x, \{equal(x, y)\})$ in our main example.

Definition 10 (Contextual equivalence rule)**.** *A **contextual equivalence rule** is a couple* $((P, \rho), T)$ *with* (P, ρ) *a contextual pattern (the **left-hand side of the rule**) and* T *a pattern over the variables of* P *and* ρ *(the **right-hand size of the rule**). We often write* $\rho \vdash (P \sim T)$.

This definition is very similar to standard equivalence rules, we simply replace the standard pattern with a contextual pattern; however there is a steep difference between how a contextual rule and a standard rule are applied. Given a contextual equivalence rule $((P, \rho), T)$ and an occurrence (X, χ, σ) of the left-hand size, applying the rule means:

- **step one** select a source n of χ and a context bearing path π for X;
- **step two** create the term n' represented by n but in which $\sigma(P)$ is replaced by $\sigma(T)$ at the end of π;
- **step three** merge the e-classes of n and n'.

The most important difference with a standard rule is that $\sigma(P)$ and $\sigma(T)$ **are not the terms made equivalent** by the rule: some source of the context and an altered version of it are.

Recall the tentative equivalence rule of Figure 27. We can now formalize it in the following way:

$$\forall x, y,\ \mathit{equal}(x, y) \vdash (x \sim y)$$

We drop the brackets around the contextual requirements before the $\vdash$ symbol. Essentially, this rule makes equivalent a source of $x = y$ with another one under which you replace a x by a y. This is the behaviour we were wishing for when we designed it in Figure 27.

Here are a few other examples of contextual equivalence rules:

- Any term divided by itself is 0, under the context that the term is not zero:

$$\forall x,\ \mathit{different}(x, 0) \vdash (\mathit{divide}(x, x) \sim 1)$$

- Under the context that $x \geq y$ and $y \geq x$, x and y are the same:

$$\forall x, y,\ \mathit{greater\_or\_equal}(x, y), \mathit{greater\_or\_equal}(y, x) \vdash (x \sim y)$$

- When we will drop the "no loop operators" simplification, we will have other new rules. For instance, under the context that c is true for the first time at a loop iteration greater than x, c is false at loop iteration x:

$$\forall c, x,\ \mathit{greater}(\mathit{pass}(c), x) \vdash (\mathit{eval}(c, x) \sim \mathit{eval}(\mathit{false}, x))$$

  This is the rule of Figure 25 as a contextual equivalence rule.

In general, any equivalence that has some form of condition can be expressed as a contextual equivalence rule.

We aim to modify the equality saturation engine to support these new contextual rules. Figure 34 shows the main loop of equality saturation improved to support contexts. We indicate in bold the new concepts and algorithms we introduce:

- the object over which the loop iterates is no longer just an EPEG but an annotated EPEG with possible context information. Section 9.2 details the precise specification of these annotations and how we can compute them on an unannotated EPEG (**Algorithm 1** on the figure);
- the left part of the figure is the standard equality saturation loop. Since we modify the EPEG when we apply rules, our possible context annotations might no longer be up to date. Rather than re-annotating the entire EPEG every time, we design an local update algorithm (**Algorithm 2**) for the annotations. This allows the annotations to be dynamically updated when the EPEG is modified. We present this algorithm in Section 9.3;

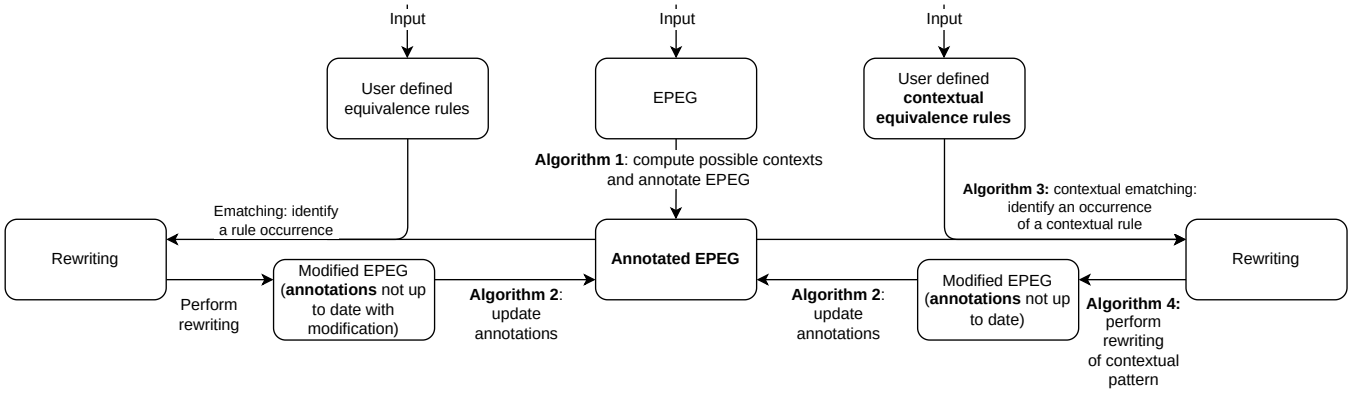


Figure 34: High-level description of our improved equality saturation engine.

- we add a new loop on the right for applying our contextual equivalence rules. This loop is similar to the standard rule application but the algorithms for finding an occurrence of a contextual rule and performing the associated rewriting differ from those for standard rules. In Section 9.4, we present **Algorithm 3** that finds occurrences of contextual patterns. In Section 9.5, we present **Algorithm 4** that performs the rewriting associated to a contextual equivalence rule.

9.2 First task: computing context annotations

Our first objective is to annotate the e-classes of the EPEG with information about their possible contexts.

A key specification of this task is that we must not only annotate each e-class X with some of its possible contexts, but also help finding sources n of the possible contexts of X (this will help with the rewriting step in Section 9.5). The way we achieve this is by attaching a "path" information to our contexts. Recall that "n is a source of possible context χ for X" if there is a certain path $\pi = (e_0, e_1, ..., e_i)$ going through the then-branches of a Φ-node per condition of χ. We enrich the context annotation with information about the last edge e_i of this path: we can find the sources of our contexts by moving upward in the EPEG, using the path information at each step. Figure 35 shows these path annotations on a simple example.

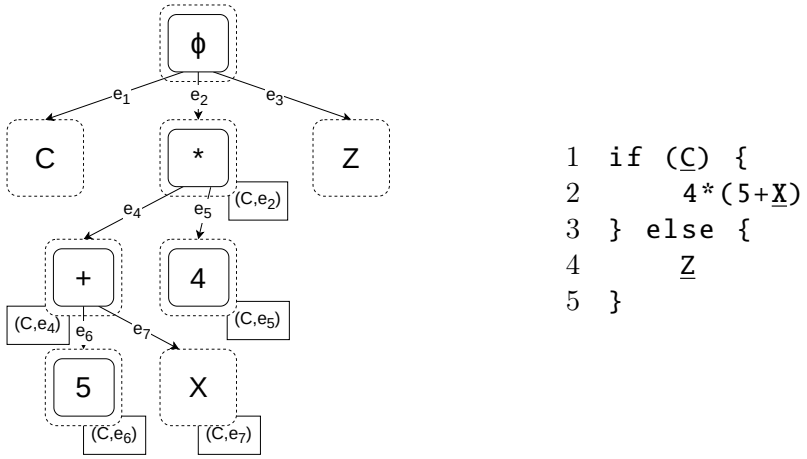


Figure 35: A simple example of PEG with context annotation and path information.

It may seem strange to record only the last edge of the path and not just the path itself. This design choice that we call **path compaction** makes sense in our case since the algorithm that uses the path information (the rewrite algorithm of Section 9.5) processes those path one edge at a time, starting from

the end: no time is gained by having access to the explicit path. Even with storage tricks to store paths in constant space (a path is always the extension of a path in a parent annotation), path compaction is a more efficient way of storing a large set of paths. This is shown in Figure 36: annotation of class X has half as many element with path compaction and we can still reconstruct the set of all context-bearing paths by moving upward.

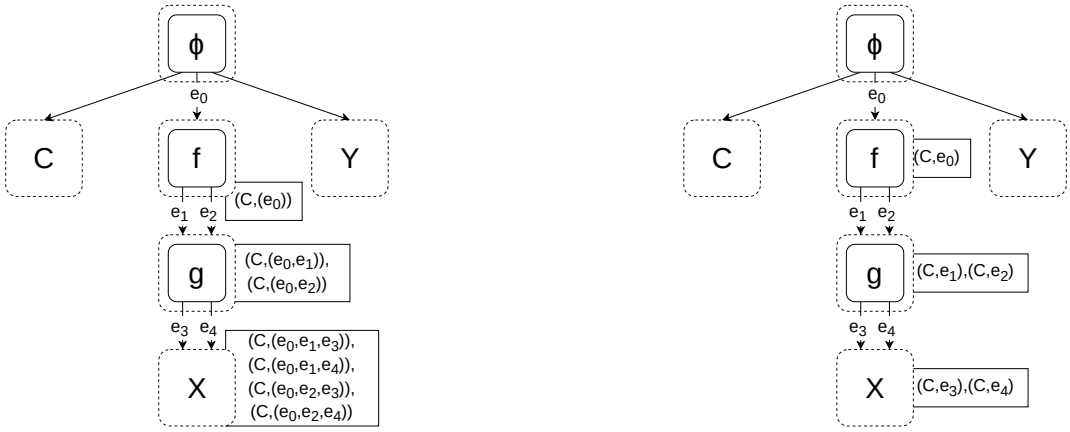


Figure 36: Comparison of the possible context annotations on a basic EPEG with explicit-path information and with last-edge path information.

Each element of our annotation now contains information about a possible context and a path. In general, different paths may lead a source n of context χ to X. We are interested in all paths for a given context since they characterize different terms for our contextual rules: in order to capture all of them, we may duplicate information about a context in an annotation if their path information is different.

Figure 37 presents some examples of how this duplication captures all paths. In *(a)*, e-class X is annotated with two couples that bear the same context: one for each direct predecessor that transmits context $\{C\}$ (even though both paths have the same source). In *(b)*, although two possible paths (with two different sources!) lead to X having possible context C, they have the same direct predecessor for X: the $*$-node. Hence only one couple in the annotation of X, but thanks to the annotation of the $*$-node, we can reconstruct that there are two context-bearing paths.

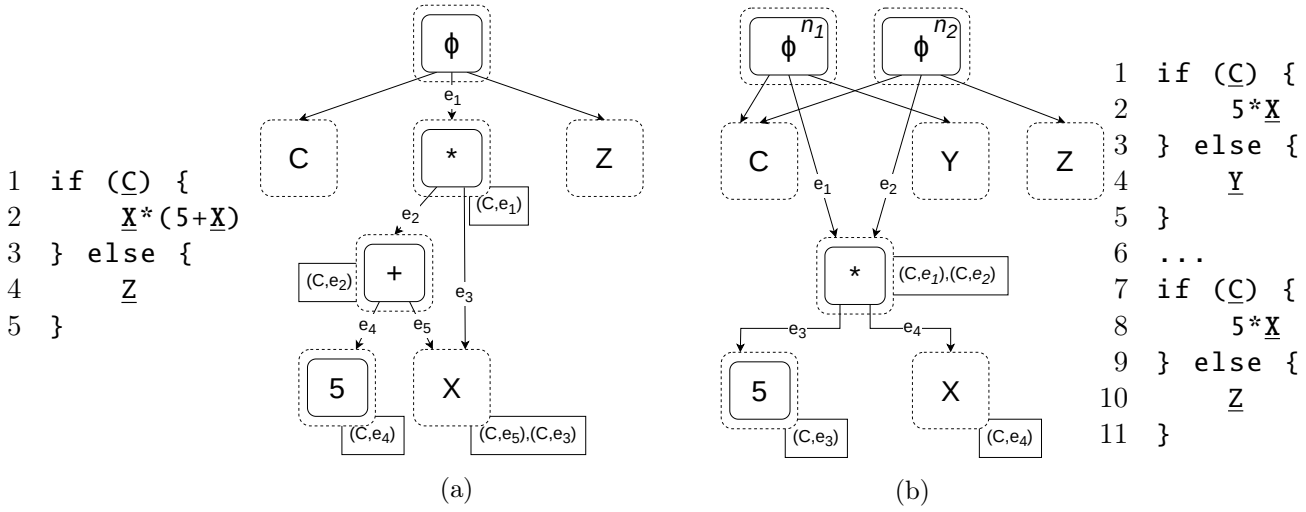


Figure 37: Cases of duplicated context information in annotations.

On the other hand, not all context are interesting: due to Lemma 1, it is sufficient to record a maximal possible context since it implies any of its subsets. Most of all, the empty context has no real use, so we

do not try to track it. Overall, we aim to under-approximate the possible contexts, and still give enough information to capture all of them and their associated paths.

Formally, the computation of possible contexts can be specified as a set constraint-based rule system. Our main variable is named $\mathcal{A}$: $\mathcal{A}$ represents the possible contexts annotation of each e-class. The annotation contains the information discussed in the previous paragraphs: for each e-class X, it is a set of couples (χ, e), with χ a possible context and e an edge leading to X transmitting this possible context.

$$\mathcal{A}(X) \subseteq 2^{\mathcal{B}} \times \mathcal{E}$$

We extend the notation $\mathcal{A}(X)$ to $\mathcal{A}(m)$ with m an e-node: an e-node has the same annotation as the e-class it is in.

$$\frac{gen(e) = \cdot}{\forall (\chi, e') \in \mathcal{A}(source(e)), (\chi, e) \in \mathcal{A}(target(e))}$$

$$\frac{gen(e) = C \quad \mathcal{A}(source(e)) \neq \emptyset}{\forall (\chi, e') \in \mathcal{A}(source(e)), (\chi \cup \{C\}, e) \in \mathcal{A}(target(e)} \qquad \frac{gen(e) = C \quad \mathcal{A}(source(e)) = \emptyset}{(\{C\}, e) \in \mathcal{A}(target(e))}$$

Figure 38: Set constraints for the specification of the possible contexts annotation. Each rule is universally quantified over the edges of the EPEG.

The set constraints specifying the computation are given in Figure 38. We comment the various rules:

- Φ-nodes are the only context-generating terms, and they only do so for their then-edges. Any edge e that does not fit this description (*i.e.* $gen(e) = \cdot$) simply transmits the possible contexts of its source to its target. The path information is updated to indicate that e is the direct predecessor in the transmission of this possible context.
- the two rules of the second line deal with the case of the then-child of a Φ. If e would transmit some contexts (first rule of the line), then they are enriched with the information that C is also, simultaneously, a possible context. Recall that we do not track the empty set as a possible context so we need the second rule in case the context annotation of $source(e)$ is empty: a new possible context must be initialized with C.

Intuitively those rules can be understood as a transitive closure of the Φ-nodes: a Φ-node is the source of a possible context that is transmitted to all its descendants.

The set constraint specification define a recursive system of inclusion rules for our annotations: the final annotation is **the smallest $\mathcal{A}$ that satisfies these rules**. Some algorithms exist to implement such this computation on a graph in general: this essentially amounts to searching a smallest fixpoint [11]. We call **analysis** the actual algorithm which computes this annotation.

The analysis of this section would be inserted at the beginning of equality saturation: right after the construction of the EPEG, it is annotated with the possible contexts. The next section explains how to dynamically update these annotations whenever the EPEG is updated.

9.3 Second task: dynamic update of the annotations

The analysis of possible contexts is a fixpoint search. It would be very costly to execute this search every time the EPEG is updated during equality saturation to update all annotations: we can actually limit ourselves to a local update. An EPEG can be modified in two ways: a new e-node can be added or two e-classes can be merged. In both cases, the update process is a downward update of the descendants of the

modified area of the EPEG: Figure 39 presents a pseudo-code for the process. This downward traversal can cease once we hit an e-class which annotation is not modified by the process. Overall the update procedure only uses local information.

The case of a new e-node n can be very simple: since n does not have any parent, the annotation of its new e-class is the empty set (no contexts). If n is not a Φ-node, it does not generate any new possible context, and transmits its empty set of contexts to its descendants: no update of the EPEG existing annotations is needed. If n is a Φ-node then the update process must be used on its *then*-child since n generates a new context for it.

In the case of a merging of e-classes X and Y, the context annotations must also be merged (paths leading to X now also lead to Y and vice versa so possible context are now shared). Since all e-nodes of the new merged e-class have potentially new possible contexts, the update process must be called on their children.

```
1  def update(e-class X):
2      old_annotation = X.annotation
3      new_annotation = X.calculate_annotation()
4      if old_annotation == new_annotation:
5          return
6      else:
7          X.annotation = new_annotation
8          for e-node n in X:
9              for child e-class Y of n:
10                 update(Y)
```

Figure 39: Update procedure pseudo-code for context annotations. The *calculate_ annotation* procedure is **local** and will be derived from our possible context analysis. It only makes use of the annotations of the direct parents e-nodes of X hence a downward transmission of the updated annotations.

We give two examples of the update process in action. In Figure 40, e-classes V and V' are merged from (a) to (b) which leads to an union of their annotations. The new annotation is transmitted to class X in (c). The update terminates due to having updated all descendants of a newly merged class, leading to the final EPEG in (d). Figure 41 shows a merge for which the annotation update only adds the information about new paths: the path information is fused for a new a new $(\{C\}, e_2), (\{C\}, e_3)$ annotation. Since $\{C\}$ was already present in the descendants annotations they are not modified (the path information only logs the direct parent which has not changed): despite an update of the parent e-class annotation, the descendants are already up-to-date.

Since the update process simply adds new information in the annotations, we can actually delay it in a similar fashion to how *egg* delayed its e-graph invariants maintenance [18]. This approach might have good performance because it would allows to do updates in batch and pool calls to *calculate_ annotation*.

We can now suppose that our EPEG is dynamically annotated with possible contexts for each e-class. In the next subsection we present an improved ematching algorithm that is able to use these annotations to detect contextual patterns.

9.4 Third task: contextual pattern ematching

There are many different ways to perform classical ematching. A naive algorithm simply traverses the EPEG looking for pattern P and tries to build a substitution σ once a structural match is found. Since slow ematching is a major time sink in equality saturation, more efficient approaches have been proposed, some of them using relational databases techniques [19]. To benefit as much as possible from these efficient algorithms, we design contextual ematching essentially as a filter over the output of a standard ematching procedure.

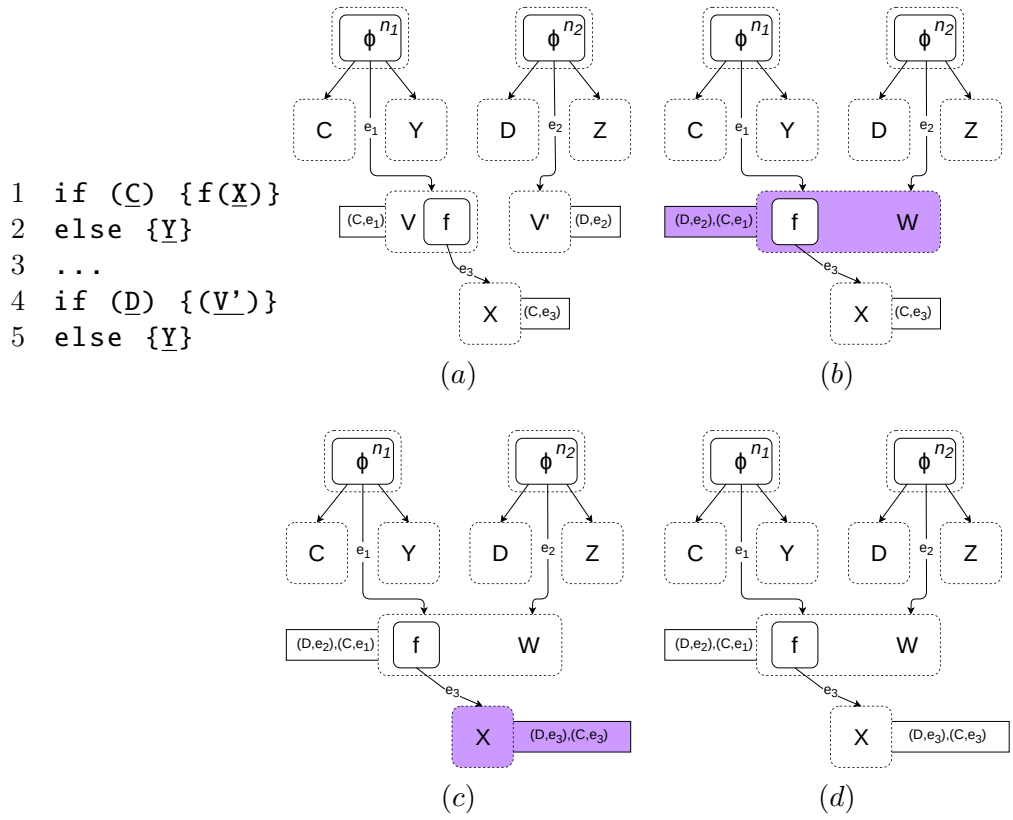


Figure 40: Step-by-step update of annotations on an EPEG.

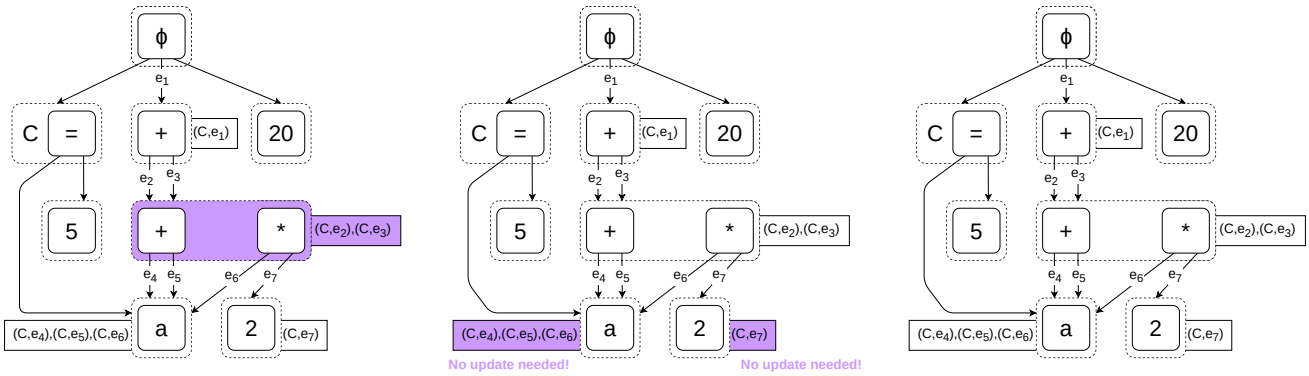


Figure 41: Merging e-classes for $a + a$ and $a * 2$ in our main example and updating annotations.

We propose the algorithm of Figure 42 to detect occurrences of a contextual pattern (P, ρ). The general idea is to first look for the standard pattern P: this leads to various occurrences (X, σ). We then filter these occurrences based on the presence of the contextual requirement $\sigma(\rho)$ in the possible contexts of X. For each occurrence of the contextual requirements in the possible contexts, we have an occurrence of the contextual pattern.

Some things of note about this algorithm:

- line 5, the contextual requirement we are looking for is $\sigma(\rho)$ and not ρ itself. This is because we are specializing the contextual requirement to the specific structural occurrence of P we found;
- line 8, we iterate over all the possible ways to match a contextual requirement within a possible context: there can indeed be more than one. For example, if the possible context contains a class

```
1  def ematching_telepattern(e-graph E, telepattern (P,rho)):
2      occurs = ematch(E,P)
3      occurs_telepattern = {}
4      for (eclass X, subst sigma) in occurs:
5          sought_context = sigma(chi)
6          for (possible_context,path) in X.annotation:
7              occurs_context = ematch_context(possible_context,sought_context)
8              for (classes,sigma2) in occurs_context:
9                  occurs_telepattern.add((X,classes,sigma2 o sigma))
10     return occurs_telepattern
```

Figure 42: Pseudo-code for contextual pattern ematching. The call to *ematch* refers to a standard ematching procedure. *ematch_context* performs a simultaneous standard ematching of the contextual requirements of *necessary_context* in *possible_context*.

with information $\{equal(W, Z)\}$ and another one $\{equal(X, Y)\}$ there is four ways to fit a contextual requirement $\{equal(x, y), equal(w, z)\}$.

- line 9, the final substitution of the occurrence is $\sigma_2 \circ \sigma$, the composition of the substitutions needed to match the structural requirement and the contextual requirements.

A somewhat advanced example of ematching is presented in Figure 43. There are three occurrences of the $(x, \{equal(x, y)\})$ contextual pattern: one in class A (substitution $x \leftarrow A, y \leftarrow Y$) and two in class M ($x \leftarrow M$ in both cases, but $y \leftarrow H$ and $y \leftarrow N$ characterize two distinct occurrences). This example shows the two main subtleties of contextual ematching that the algorithm has to take into account:

- contextual requirements must be specified for a given occurrence of P: the first occurrence has context in which x is A, the two other contexts in which x is M.
- given an occurrence of P in e-class X, different possible contexts of X may satisfy the contextual requirements: the two last occurrences focus on the same occurrence of the standard pattern, in M, but with two different contexts.

```
1  if (A=B) {A}
2  else {X}
3  ...
4  if (M=H) {M}
5  else {Y}
6  ...
7  if (M=N) {M}
8  else {Z}
```

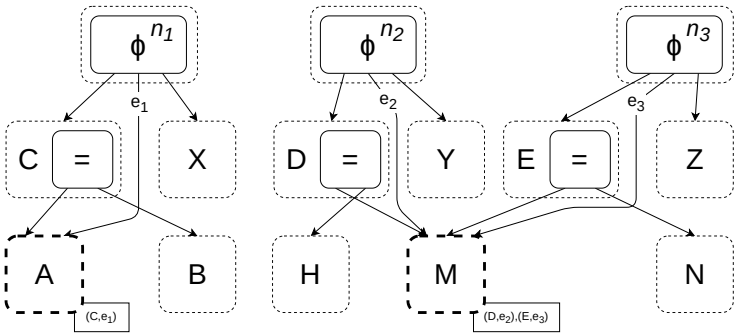


Figure 43: A PEG in which we ematch the contextual pattern $(x, \{equal(x, y)\})$.

The contextual ematching process we defined here can now be applied to contextual rewriting rules. The next subsection gives an algorithm to perform rewriting.

9.5 Fourth task: rewriting for contextual equivalence rules

We recall the three steps for applying a contextual equivalence rule. Given a contextual equivalence rule $\rho \vdash (P, T)$ and an occurrence (X, χ, σ) of the left-hand side, applying the rule means:

- **step one** select a source n of χ and a context bearing path π for X;
- **step two** create the term n' represented by n but in which $\sigma(P)$ is replaced by $\sigma(T)$ at the end of π;
- **step three** merge the e-classes of n and n'.

Note that given a rule and an occurrence of its left-hand size there are possibly many different ways to do step one: the various sources of χ and the various context-bearing paths from these sources to $\sigma(P)$ can be considered for the rewriting. This is another difference with the standard equivalence rule definition. We design our rewriting algorithm so that it does the rewrite for every possible source and every **acyclical** context-bearing path (cycles can sometimes happen in the EPEG of a loopless program due to e-class merging).

```
1  REWRITE_TELEPATTERN(e-graph E, occurrence (X,chi,sigma), rule ((P,rho),T))
2      term new_t = sigma(T)
3      e-class X0 = E.add(new_t)
4      tokens M = [not_visited for every e-node of E]
5      AUX(X0,X,chi,M)
6
7  AUX(e-class X, e-class X', context chi, tokens M)
8      for each (chi',e) in X'.annotation such that chi subseteq chi':
9          n = source(e)
10         if n visited in M:
11             break
12         M' = M where n is now visited
13         new_t = n but replace e with an edge to X
14         new_X = E.add(new_t)
15         if cond(e) in chi:
16             new_chi = chi \ {cond(e)}
17         if new_chi is empty:
18             merge(new_X,n.class)
19         else:
20             AUX(new_X,n.class,new_chi,M')
```

Figure 44: Pseudo-code for the contextual equivalence rule rewriting procedure.

A pseudo-code for the rewriting algorithm is presented in Figure 44. The general idea is to create the class X_0 corresponding to term $\sigma(T)$ and then from X, move upward the context-bearing paths in the e-graph while building the same structures above X_0. Some commentaries about specific algorithmic details:

- line 4, we initialize a table of visit tokens. These tokens are used to avoid considering cyclical paths and guarantee the termination of the process.
- line 7, procedure AUX has two e-classes parameters. X is the e-class at the stem of the branch being constructed (which we will call the **bough** here) with the new pattern $\sigma(T)$ at its end, X' is the corresponding e-class in the already existing branch with old pattern $\sigma(P)$ at its end. The other parameters are the context requirements χ and the visit tokens M.
- lines 13 and 14 are the main engine of the bough construction. The structure of n, a parent of X' in a context-bearing path is copied above the bough X. The new bough stem is now the class of this recreated structure.

- lines 15 and 16, the context requirements χ are updated whenever an edge which condition is in χ is met. This allows to track which context requirements have not yet been met in the bough.
- lines 17 and 18, having met all our contexts requirements means we have reached a source for our context requirements. The source of the bough can be merged with the source of the context-bearing path.

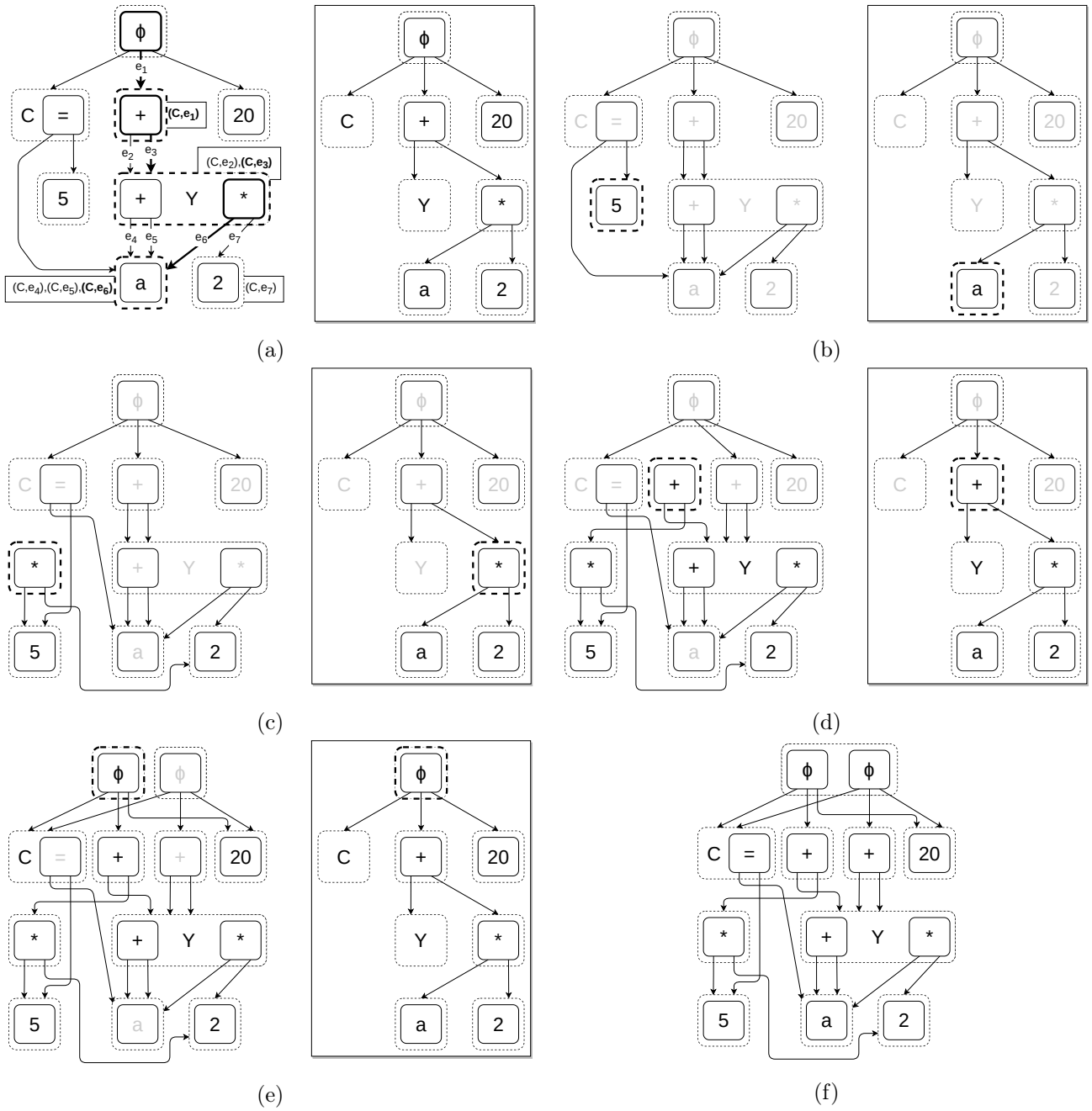


Figure 45: Step-by-step modifications of our main example EPEG by the rewriting process with equivalence rule of Figure 27.

We show a partial execution of this process on our main example in Figure 45 using the aforementioned rule $equal(x, y) \vdash (x \sim y)$ with occurrence spotted in Figure 33: note that there are six possible context bearing paths for this occurrence of the pattern, but we only show the rewriting for one of them: (e_1, e_3, e_6).

The box shows the structure of the branch being copied (although the algorithm normally discovers this structure during its upward traversal of the branch). Step-by-step:

- in *(a)*, the initial EPEG, in bold the context-bearing path from the Φ-node to the a-node and in the box the isolated structure of this branch. This is the structure we are going to copy with the bough.
- in *(b)*, we initialize the bough with an e-node 5 as the rule states that a is to be replaced by 5. The e-class of 5 already exists in the EPEG so this is where we begin.
- in *(c)*, we move upward in the branch: right above the class of a is a $*$-node with the class of 2 as a second child. We copy this structure above the bough, leading to a new e-class in the EPEG for term $5 * 2$.
- in *(d)*, we continue to extend the bough, now with a $+$-node which first child is Y as the branch dictates.
- in *(e)* another extension with a Φ-node of condition C and else-branch the class of 20. The Φ-node is a source of $\{C\}$ in the branch, the extension process ends here.
- in *(f)*, the e-classes of the bough and of the source of the path are merged. This is the final EPEG.

A complete execution of the algorithm would create six new Φ-nodes from which, at the end of a context-bearing path, an instance of a has been replaced by an instance of 5. In our example case, notice that if we were to substitute a by 5 at the end of all six paths *at the same time*, some major constant folding could happen in the then-branch of the Φ-node. What we did with the rewriting process does not immediately allow this constant folding, some more applications of the contextual equivalence rule are needed in order to get a Φ-node under which all instances of a in the then-branch are replaced by 5. Still, this is better than extremely high number of swapping rules needed to perform this with standard rules.

Our algorithm notices a set of possible rewrites and does each one of them separately. For our example, it would be more advantageous to immediately combine all possible rewrites into one term but this might not always be the case. Our approach allows us to access all intermediary terms in which only some of the rewriting were done, which might sometimes open new options for equality saturation.

10 Discussion of our new algorithmic framework

10.1 Generalization to all EPEGs

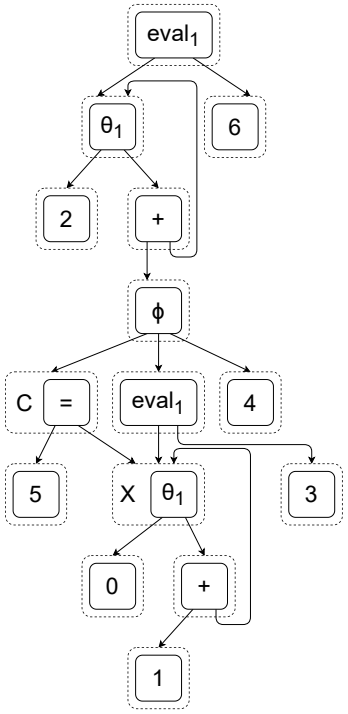
In this subsection we drop the two simplifications we made in the report: loop operators are now allowed in PEGs and nested loop structures in program. Without too much detail, PEGs for programs with nested loops parameterize their loop operators with identifiers from a set of loop identifiers $\mathcal{L}$ (so instead of just $pass, eval$ and θ, we would have $eval_l, eval_{l'}, pass_l, pass_{l'}, \theta_l, \theta_{l'}$ for instance with two loop identifiers l and l'). The semantics of terms is no longer parameterized by the current loop iteration i but by a loop iteration index ι that gives the current iteration for all loop depths.

Our algorithmic framework can be generalized to include all EPEGs but this requires considering a few subtleties. In this subsection we give a high-level description of how this modifies our algorithms.

Context propagation: the semantics of the terms in an EPEG is parameterized by the current loop iteration index ι. Without any loop operator, we were free to ignore this fact: all terms are computed at the same loop iteration index ι since there is no loop. This means that if an e-class Z is on the then-branch of a Φ-node $\Phi(C, X, Y)$, condition C necessarily holds if Z is computed since both are computed at index ι and C was true.

With loop operators, this is no longer true. Loop operators rely on changing the loop iteration index for their computation and this change might invalidate some possible contexts acquired above in the EPEG. For instance, take the EPEG of Figure 46. Both loops of the program are indexed as "loop 1" as they

are not nested: this is the standard loop indexing of Peggy. In this EPEG, the Φ-node is not a source of context C for e-class X, despite X being on the then-branch. This is because, on the then-branch, X is computed at iteration 3 while C is true at iteration 5.




```
1  int x = 0
2  repeat 3 times {
3      x = x  + 1
4  }
5  int y = 2
6  int z = 0
7  repeat 6 times {
8      if (z == 5) {
9          y = y + x
10     } else {
11         y = y + 4
12     }
13     z = z + 1
14 }
15 return y
```

Figure 46: An EPEG with a loop operators and associated program.

A way to deal with this phenomenon is to modify modify *cond* so that some edges generate possible contexts (the then-children of Φ-nodes) and some others annihilate possible contexts (every edge of a loop operator that relies on changing the loop iteration index). Possible contexts are not transmitted through those edges. This requires altering the algorithms for computing and updating the context annotations.

A new source of possible contexts: once we add the loop function symbols, we can also consider a new type of context-generating structure, not just the Φ-nodes. The loop body X of a term $eval(X, pass(C))$ is evaluated at a loop iteration index at which C is true. The *eval*-node can be seen as a source of possible context C for X.

The specifications of our algorithms are general enough to allows for new sources of possible contexts, but there is a subtlety here: the context source relies on two different nodes rather than just one. The *eval*-node does not generate context on its own, it needs a *pass*-node as a child to do so. This complexifies the update algorithm as it needs to track new *eval-pass* structures being created by modifications of the EPEG. The process remains local as we only need to check the children and parents of the modificated e-class, a variation of the algorithm we proposed can work.

Overall, the algorithmic framework we defined can be generalized to all EPEGs.

10.2 Possible extensions

Taking in consideration false contexts: it seems natural to track not only possible contexts that are known to be true but also possible contexts that are known to be false. The framework we presented can be extended by attaching a boolean to possible contexts and adding that Φ-nodes now generate their condition as a false context for their else-child.

This feature can be somewhat emulated with true possible contexts and the use of an equivalence rule that flips Φ-nodes: $\Phi(C, X, Y)$ is equivalent to $\Phi(\neg C, Y, X)$. Whenever a rule would require C being false,

it now instead requires $\neg C$ being true. This approach is however less elegant, forces a specific equivalence rule to be used and increases the contextual requirements term size.

Taking in consideration universally true contexts: any e-node in the same e-class as the *true*-node represents terms that are true regardless of loop iteration or program inputs. These universally true terms can be seen as "free context": we could exploit them to satisfy some contextual requirements of our contextual equivalence rules.

Smarter propagation through loop operators: since some edges of loop operators might invalidate some possible contexts, we decided in Section 10.1 to annihilate all contexts passing through those edges. This is a conservative approach, some smarter policies exist. For example, we could spare possible contexts that are invariant of the loop of the operator. For example, if C is invariant of loop l, then we could allow C to be transmitted through a $pass_l$.

Other analyses as contexts: we can piggyback our analysis to propagate other kinds of information in an EPEG. For example knowing that a term in the EPEG is computed at a given loop iteration index can allow for some transformations.

We could generalize our algorithmic framework to support other analyses in the same way egg has the e-class analysis framework [18]. The two approaches seem to support different kind of analyses: notably, an e-class analysis is propagated from children to parents in the EPEG, while our context analysis is transmitted from parents to children.

Conclusion

During this internship we studied translation validation for the Java to Kotlin conversion. We reviewed a variety of Java features: how they are translated by the IntelliJ Java to Kotlin converter and how they are compiled by the Java and the Kotlin compilers. We also confronted the Peggy equivalence checker tool to this specific problem and improved its expressivity, allowing it to catch new kinds of equivalences. Lastly, we proposed a new algorithmic framework that would allow the equality saturation approach to support a new kind of rules: contextual equivalence rules.

This work was the occasion to try the equality saturation technique in the specific case of Java to Kotlin conversion. The approach is both powerful, as it is able to prove some non-trivial equivalences and very generic as it could be adapted to our problem without modifying the core algorithmic engine. Of course, a solver-like technique for an undecidable problem such as program equivalence is bound to face incompleteness in the form of timeout or insufficient axiom sets. Our contextual-rewriting contribution would allow to design more advanced axioms sets. The new algorithmic framework we propose would hopefully push back the timeout boundary by improving performance.

An improvement of this work would be to prove the semantic soundness of our algorithms and concepts. Much like there are sound equivalence rules, we would also define what is a sound contextual equivalence rule. It is important that the EPEG itself remains sound through calls to our rewrite algorithm, *i.e.* always represents equivalence classes of terms that are indeed semantically equivalent. The high-level form of a soundness theorem for the whole framework would therefore be: "if a contextual equivalence rule is sound, then using the rewrite algorithm on an occurrence of the rule found by the contextual ematching algorithm produces a new sound and enriched EPEG".

Another further work would be to implement and experiment with our new framework, either in an existing tool like *egg* or in a new tool specific to program equivalence checking. During the internship we put the focus on the algorithmic part of the equality saturation engine. The implementation of a scalable equivalence checker would require additional and specific efforts. For instance the bytecode to PEG conversion requires advanced SSA-like compiler transformations. Equality saturation is also suitable for parallelization and we would like to add this feature to our algorithmic framework.

References

[1] Intellij website. https://www.jetbrains.com/idea/. Accessed: 2023-06-01.

[2] Kotlin documentation, comparison to java. https://kotlinlang.org/docs/comparison-to-java.html#some-java-issues-addressed-in-kotlin. Accessed: 2023-06-01.

[3] Peggy website. https://goto.ucsd.edu/~mstepp/peggy/. Accessed: 2023-06-08.

[4] F. Bannwart and P. Müller. A program logic for bytecode. *ENTCS*, 141(1):255–273, 2005. in Proc. of Bytecode 2005.

[5] G. Barthe, S. Blazy, V. Laporte, D. Pichardie, and A. Trieu. Verified translation validation of static analyses. In *2017 IEEE 30th CSF*, pages 405–419, 2017.

[6] Gilles Barthe. An introduction to relational program verification. https://software.imdea.org/~gbarthe/__introrelver.pdf, 2020. Accessed: 2023-06-01.

[7] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proc. of the 31st ACM SIGPLAN-SIGACT*, POPL '04, page 14–25, 2004.

[8] N. Benton. Semantic equivalence checking for hhvm bytecode. In *Proc. of 20th PPDP*. ACM, 2018.

[9] Peter J. Downey, Ravi Sethi, and R. Endre Tarjan. Variations on the common subexpression problem. *J. ACM*, 27(4):758–771, oct 1980.

[10] S. Freund and J. Mitchell. A type system for the java bytecode language and verifier. *Journal of Automated Reasoning*, 30, 12 2003.

[11] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer Publishing Company, Incorporated, 2010.

[12] R. Nielson and F. Nielson. *Semantics with Applications: An Appetizer*. Undergraduate Topics in Computer Science. Springer, March 2007.

[13] Michael Sipser. *Introduction to the Theory of Computation*. Course Technology, Boston, MA, third edition, 2013.

[14] M. Stepp. Equality saturation : engineering challenges and applications, 2011. Thesis at UC San Diego.

[15] M. Stepp, R. Tate, and S. Lerner. Equality-based translation validator for LLVM. In *Computer Aided Verification*, pages 737–742, 2011.

[16] R. Tate. Equality saturation : using equational reasoning to optimize imperative functions., 2012. Thesis at UC San Diego.

[17] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: A new approach to optimization. *LMCS*, 7, mar 2011.

[18] M. Willsey, C. Nandi, Y. Wang, O. Flatt, Z. Tatlock, and P. Panchekha. egg: Fast and extensible equality saturation. *In Proc. of the ACM on Programming Languages*, 5:1–29, jan 2021.

[19] Y. Zhang, Y. R. Wang, M. Willsey, and Z. Tatlock. Relational e-matching. *Proc. ACM Program. Lang.*, 6(POPL), jan 2022.

Annex

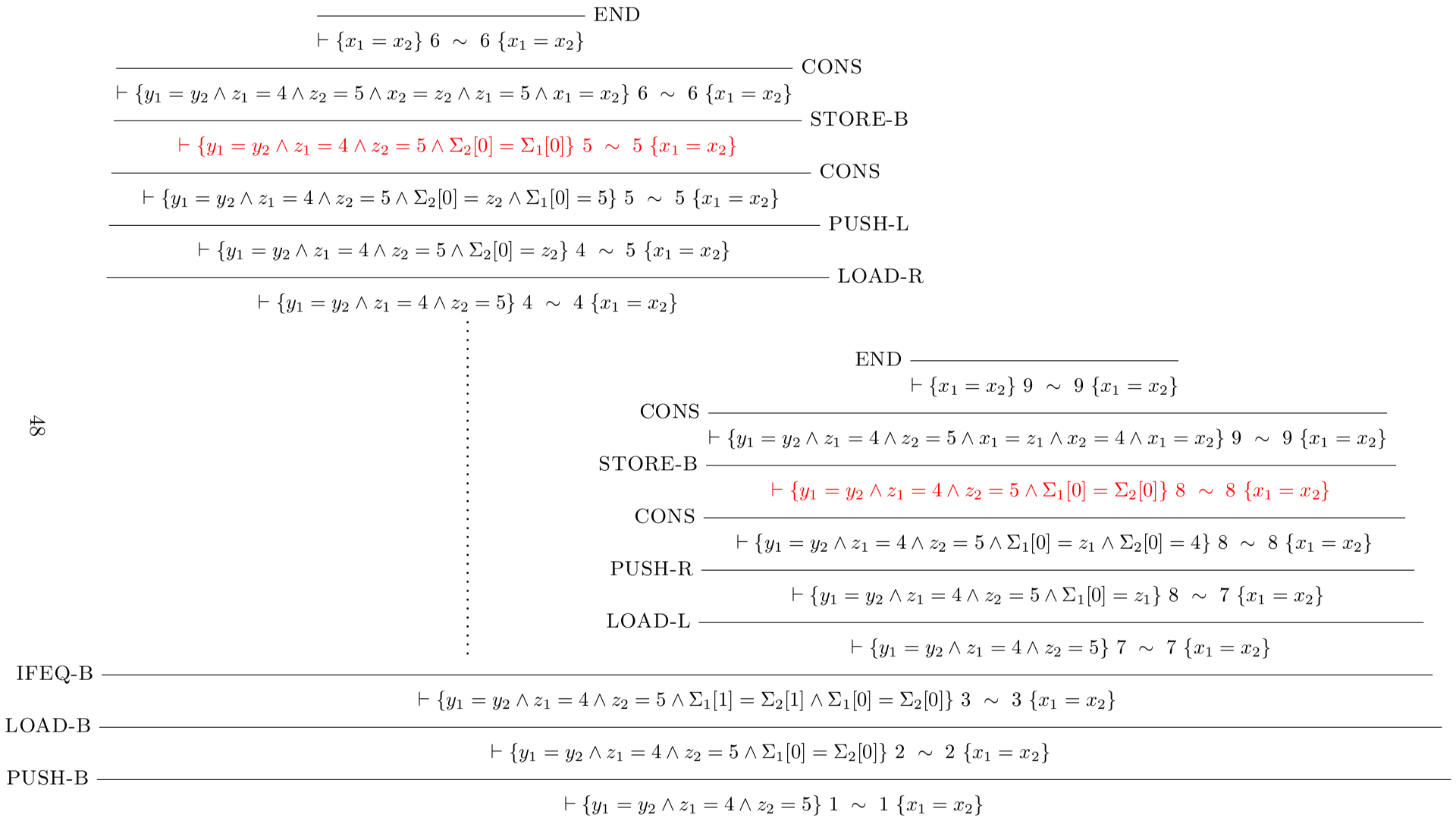


Figure 47: A proof tree for the statement $\vdash \{y_1 = y_2 \wedge z_1 = 4 \wedge z_2 = 5\}\ 1\ \sim\ 1\ \{x_1 = x_2\}$ about the bytecodes of Figure 3

Class main feature	**Observed difference**	**Can Peggy prove equivalence?**
Arithmetic operations	Permutation of registers used and optimization	Yes
Basic constant folding	Permutation of registers used and optimization	Yes
Conditional structure	Expression-if	Yes
Constant foldable conditional	Optimization: removed conditional dead branch	Yes
Advanced dead code (repeated if)	Expression-if	Yes
Boolean and conditional	Expression-if	Yes
Basic loop for on *int*	Different control flow and loop exit condition	No
Loop while	Permutation of registers used	Yes
Nested while loops	Permutation of registers used	Yes
Increment-by-two loop for	Converted into a while, permutation of registers	Yes
Switch	Expression-switch	Yes
Recursion	Expression-if	Yes
Terminal recursion	Expression-if	Yes
Infinite while-loop	Kotlin bytecode has no return instruction	No
Calling another function	Optimization: suppressed an useless local variable	Yes
Creating object, accessing field	Field access is a method in Kotlin	No
Uninitialized local/field	Kotlin: all fields must be init, all locals implicitly init	Yes
Creating and accessing array	None	Yes
Iterating over array	Permutation of registers used and some operations order	Yes

Figure 48: Observed differences in basic Java and Kotlin programs.

Rule name	**Variables**	**Leftside pattern**	**Rightside pattern**
Φ-reversing	c, x, y	$\Phi(c, x, y)$	$\Phi(\neg(c), y, x)$
Φ-swapping	c_1, c_2, x, y, z	$\Phi(c_1, \Phi(c_2, x, y), z)$	$\Phi(c_2, \Phi(c_1, x, z), \Phi(c_1, y, z))$
Φ-or expansion	c_1, c_2, x, y	$\Phi(or(c_1, c_2), x, y)$	$\Phi(c_1, x, \Phi(c_2, x, y))$
Φ-and expansion	c_1, c_2, x, y	$\Phi(and(c_1, c_2), x, y)$	$\Phi(c_1, \Phi(c_2, x, y), y)$
eval-Φ left distribution	c, x, y, z	$eval(\Phi(c, x, y), z)$	$\Phi(eval(c, z), eval(x, z), eval(y, z))$
eval-Φ right distribution	c, x, y, z	$eval(z\Phi(c, x, y))$	$\Phi(c, eval(z, x), eval(z, y))$
Φ-solving by *pass*	c, x, y	$eval(\Phi(c, x, y), pass(c))$	$eval(x, pass(c))$
eval at successor	x, y	$eval(x, succ(y))$	$eval(shift(x), y)$
pass is positive	c	$gt(0, pass(x))$	$false$
pass-or is a minimum	c, d	$pass(or(c, d))$	$\Phi(gt(pass(c), pass(d)), pass(d), pass(c))$
gt-destruction	x, y	$gt(x, y)$	$and(\neg(equal(x, y)), \neg(lt(x, y)))$
gt successor (right)	x, y	$gt(x, succ(y))$	$and(gt(x, y), \neg(x, succ(y)))$
gt successor (left)	x, y	$gt(succ(x), y)$	$or(equal(x, y), gt(x, y))$
Java-Kotlin loop *for*	i_0, i_f	$pass(gt(i, i_f))$ where $i = \theta(i_0, plus(1, i))$	$\Phi(gt(i_0, i_f), 0, succ(pass(equal(i, i_f))))$
Use Φ-equality	x, y, z	$\Phi(equal(x, y), x, z)$	$\Phi(equal(x, y), y, z)$
Use Φ inequality on *pass*	c, w, z	$\Phi(gt(pass(c), w), eval(c, w), z)$	$\Phi(gt(pass(c), w), eval(false, w), z)$

Figure 49: Rules used in the loop-for with break equivalence proof.