A Framework for Bidirectional Program Transformations

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A Framework for Bidirectional Program Transformations

Master's Thesis

Simon Fritsche

April 20, 2017

Advisors: Dr. Malte Schwerhoff and Prof. Dr. Peter Müller

Chair of Programming Methodology
Department of Computer Science, ETH Zurich

Contents

Chapter 1

Introduction

Many people see programs only as text that performs work on their computer after pressing the "run" button in the development environment. But when working in the area of compilers, program analysis or program verification, a program is not only characters in a file but an object in memory to work with. When working with programs in such a context, they are usually represented as a tree data structure called *Abstract Syntax Tree* or in short *AST*.
Processing of programs (e.g. translation into machine code, optimization, verification) is far easier on a program in AST form than in textual form.

The main goal of this thesis is to provide a framework that makes the encoding of program modifications on AST level as convenient as possible. Since this framework is for modifying, i.e. rewriting ASTs, we will call it a *rewriter* throughout this thesis.
We want to provide a simple way of specifying standard use cases that is still able to provide the expressiveness which is required for more complex use cases. The rewriter should be usable on arbitrary tree structures and even on *Control Flow Graphs* that include cycles. It should provide a second layer language that improves the specification of AST transformations. We want bidirectional transformations, which means that we want to be able to transform selected parts of a program back into the state where no transformations were applied yet. This is useful, for example, in the context of error messages. A more in depth specification of all the goals we set and achieve during this thesis is provided in Section 1.3.

The proposal for developing a rewriter came from the Chair of Programming Methodology at ETH Zurich. They want to have such a framework for the development of their program verification infrastructure called *Viper*. More information on Viper is provided in section 1.2.
There was already a rewriter implementation present in the Viper infrastructure but its usability and functionality is limited. We call this framework the

existing rewriter. In subsequent chapters of this thesis, we will point out the limitations of the existing rewriter and how we overcame them in the new rewriter.

We also looked into existing work regarding AST transformation and extracted useful ideas from them. The rewriting frameworks we looked at are: The Stratego [1] program transformation language, Kiama [11] library for language processing. We looked at *Tregex* [13] to get ideas for our DSL. The masters thesis of Leo Büttiker [3] also provided useful input for our rewriting framework.

The following sections of this chapter explain the concepts that are required in order to understand the rest of this thesis.

1.1 Abstract Syntax Trees

As already mentioned, ASTs are the go-to data structure for representing programs. This thesis is all about modifying and rewriting these trees.
Figure 1.1 shows how assignment `x := 1 * y + y / 2` looks in AST form, where `x` and `y` are integer variables and `:=` is the assignment operator.

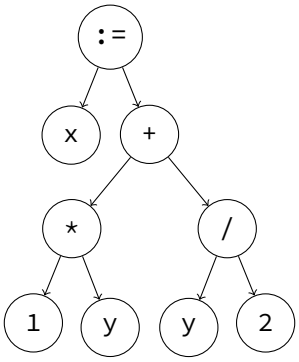

Figure 1.1: AST representation of the assignment `x := 1 * y + y / 2`

1.1.1 Terminology

Here we explain the different names we use for groups of elements in the AST. Figure 1.2 shows an AST where elements in the same group are marked with the same letter.

The element groups are:

- *Node* (N): N is the node we are currently looking at. All the other groups of elements only make sense relative to a certain node. We call N the *current node*.

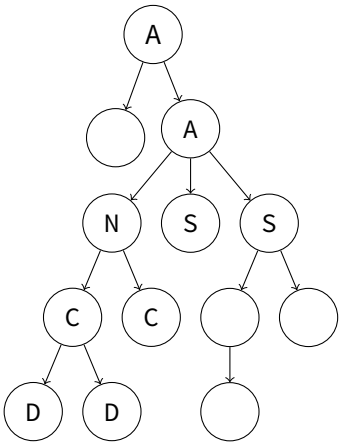

Figure 1.2: Node groups in an AST

- *Children* (C): Children are the direct successors of the current node in the tree. Every node in an AST holds information about its direct children.
- *Ancestors* (A): Ancestors are the nodes that form the path from the *root* of the tree to the current node (N). The first ancestor is the root. The last ancestor is the current node itself and the second to last ancestor is the node directly above. the current node and is called *parent*.
- *Siblings* (S): Siblings are nodes that share the same parent. The list of siblings does not include the current node.
- *Descendants* (D): Descendants are nodes that have the current node as ancestor. Note that this includes the children (C) of a node.

1.2 Viper

The Verification Infrastructure for Permission-based Reasoning (Viper [6]) is a collection of tools developed by the Chair of Programming Methodology group at ETH Zurich.

The Viper infrastructure includes:

- Viper intermediate language: an intermediate verification language based on a permission logic inspired by Implicit Dynamic Frames [12].
- Frontends that translate higher level programming languages (Python, Java, Scala, etc.) into the Viper language
- Backends that analyze or verify the Viper code

The core goal of the Viper tool-chain is to make the implementation of program verification easier. Viper gives easy access to verification by providing

an intermediate layer between theorem provers and high level code. To verify a programming language, one can simply implement a translation from the verification techniques of said programming language into Viper code.

The complete Viper infrastructure is implemented in Scala. Therefore we chose to implement the rewriter in Scala as well.

1.3 Goals

This section describes the goals we achieved in this thesis alongside references to the chapters that tackle these Goals.

1.3.1 Rewriting Core

These goals describe the functionality we want for the new rewriting framework. The solutions that achieve these goals are explained in chapter 3.

1. Common transformations that were possible to implement with the existing rewriter should be possible to implement with the new rewriting framework. Furthermore there should be no additional specification overhead generated by the new rewriting framework.
2. The implementation of transformations should be enhanced by making important information about the current node available through the rewriting framework. This means access to: ancestors, siblings and even special user defined information.
3. Children selection and AST node duplication should be treated separately (not the case in the existing rewriter). We want this modularity because it gives us the possibility to optimize and automate children selection and duplication separately.
4. The new rewriter should provide a feature to control the recursion into AST nodes on node level and on groups of nodes.
5. The rewriting framework should be able to transform arbitrary ASTs not only the Viper AST. Furthermore we want to be able to deal with cycles in order to transform CFGs.
6. We want to be able to combine transformations. Complex AST transformations can be the combination of multiple simple transformations.

1.3.2 Error Transformations

These goals describe what we want for the other direction of the bi-directional transformation. We call these transformations into the untransformed state of the program *back-transformations*. The concept of back-transformations is

implemented with the example of error message back-transformations. How we achieve these goals is described in chapter 4.

7. The rewriter should provide the ability to transform nodes back into the untransformed state without additional help of the user
8. The user should be able to select individual nodes, not necessarily the whole AST, and transform them back.
9. We want back-transformations to be integrated into the Viper error message framework to map error messages back into the user context (example in 2.5).

1.3.3 DSL

The goals in this section describe what we expect from the second-layer DSL and what the purpose for the design was. These goals are achieved in chapter 5. Further explanation about the DSL we chose can be found in section 1.6

10. The DSL should provide a powerful and expressive language to specify transformations. We want the DSL to address the problem of deep matching (matching on parts of the AST that are deeper down than the current node, motivated in Section 2.7).
11. Although the DSL should be designed to be powerful and flexible, standard cases have to be implementable in a simple and concise way.

1.4 Scala

As we already mentioned, the rewriting framework is implemented in Scala because it compliments the Viper infrastructure, which is implemented in Scala. Therefore, a majority of the code examples in this thesis are written in Scala. This section gives a brief introduction into the Scala features that are used in this thesis but it is not a complete tutorial for Scala.
If one is interested in Scala or needs more information to understand a code example, there is a comprehensive step-by-step guide from the author of Scala: Martin Odersky [7].

1.4.1 Variables and Collections

Scala *vals* are equivalent to constants or final variables/fields in other common programming languages such as Java, C, etc. Since Scala includes a lot of features from functional programming languages, immutable variables and immutable data structures are more strongly advocated, compared to most other object oriented languages. In fact, one should always declare a

variable as a `val` if possible. If a variable has to be mutable, one can use the `var` keyword.

All data structures in Scala are immutable by default. If we want to add an element to a list we have to create a new immutable list. To model a mutable list with an immutable list, we use a mutable variable (`var`) and always reassign the new list to the variable. Listing 1.1 shows the use of `vals`, `vars` and `Lists`. The `++` operator is used to append one list to the other.

```
var part1 = List(1, 2, 3)
val part2 = List(4, 5, 6)
part1 = part1 ++ part2
// part1 is now List(1, 2, 3, 4, 5, 6)
```

Listing 1.1: Scala vals,vars and immutable lists in use

Lists also support a lot of useful functions for modification such as `map`, `filter`, `collect`, etc. If examples including those functions occur in the thesis and are not understood, consult the Scala documentation website [9].

1.4.2 Case Classes

Case classes provide convenient additional features compared to regular classes. The concept of a case class is that the class and the behavior of a class is defined by a core set of fields. They are Scala's way of defining Abstract Data Types (ADTs). An example of a case class is provided in listing 1.2.

```
case class IntLit(i: Int)
...
val lit = IntLit(5)
val value = lit.i
```

Listing 1.2: Example of a case class

Class `IntLit` wraps an integer in a literal class. The parameter of the case class (`i: Int`) behaves like a field. The idea behind case classes is that these field(s) define the case class. Scala provides features for case classes based on this assumption:

- Case classes provide a default implementation for equals and hash code that include all the fields. This is called structural equality.
- Case classes support pattern matching (explained in section 1.4.3)
- Case classes implement the `Product` trait. `Product` is the base trait for classes that are defined by a number of elements e.g. tuples, functions, case classes. For case classes the product elements are its fields. Every subclass of the `Product` trait has to implement an abstract function

that allows access to the individual product elements. This will prove useful to access the fields of a case class in Section 3.2.3.

- Case classes do not need the `new` keyword for instantiation. This is important to have in mind when reading Scala code.

1.4.3 Pattern Matching

Scala pattern matching provides a control structure similar to a switch statement. Assume we have the two case classes from Listing 1.3:

```
trait Expression

case class IntLit(i: Int)
  extends Expression

case class BoolLit(b: Boolean)
  extends Expression
```

Listing 1.3: Example class hierarchy for boolean and integer literals

With pattern matching we can now find out if a variable of type `Expression` is an `IntLit` or a `BoolLit`. We can match depending on the wrapped integer or boolean value and we can provide custom predicates that have to hold on a match. Listing 1.4 demonstrates some of the possibilities.

```
val exp:Expression = ...

val abs = exp match {
    case BoolLit(true) => 1
    case IntLit(x) if x < 0 => -x
    case IntLit(x) => x
    _ => 0
}
```

Listing 1.4: Example of pattern matching

The control flow of the program will go from top to bottom through the `case` statements. If a case matches the matched expression (`exp`), the expression on the right hand side of the `=>` is evaluated and taken as the return value of the pattern match (`match` keyword).
Note that local variables can be defined and bound to the value of a field, as we see in the case of `case IntLit(x) => x`. On the left side `x` is defined to be the parameter of the `IntLit` class and on the right side `x` can be used as variable that denotes the value of field `i` of class `IntLit`.
Scala also defines the _ pattern that matches everything and is mostly used as a default case. If nothing matches inside the `match` block, an exception is thrown.

1.4.4 Functions and Methods

Listing 1.5 shows the basic skeleton of a Scala method.

```
def meth(param: PType): RType = body
```

Listing 1.5: Sketch of a Scala method

A method is defined with the `def` keyword. `meth` takes one parameter with name `param` of type `PType`. The return type is stated immediately afterwards, in this case it is `RType`. `body` describes the function body. The body can be a statement block, but can also be just an expression.

In Scala every statement is also an expression. This means that every statement defines a return value. The return value of a statement block is the return value of the last expression that was evaluated. Listing 1.6 shows two examples. If a statement is not supposed to return anything (e.g. for-loop) then its return value is the `Unit` value, comparable to `void` in other programming languages. For more information look at the book "Programming in Scala" [7].

```
val two = {
  val x = 4;
  x-2
}

val three = if(two == 2) {
  3
} else {
  2
}
```

Listing 1.6: The return value of each statement is the name of the `val` it is assigned to

1.4.5 Anonymous Functions

Functions can be anonymous in Scala. This means that one can declare functions without a name. Combined with the concept that functions are first-class objects i.e. they can be assigned to a variable or passed around as a parameter, anonymous functions are a powerful feature. Examples for anonymous functions can be seen in listing 1.7.

```
val add = (x:Int, y:Int) => x + y

val combine = (x:Int, y:Int,
  comb:Function2[Int, Int, Int]) => comb(x,y)

val five = combine(2, 3, add)
```

Listing 1.7: Examples of anonymous functions. Before the => is a parameter list and after the => is the body.

Variable add is a reference to an anonymous addition function. combine references a function with three parameters. The third parameter of combine has to be of type Function2[Int, Int, Int]. This means that the actual argument provided as the third parameter of combine has to be a function that takes two Ints as parameters and returns an Int. Function combine then applies parameter x and y to the comb parameter.
Since the function referenced by add is such a Function2 we can, for example, add 2 and 3 together by calling combine with 2, 3 and add as parameters.

A more concise way to specify anonymous functions is to use the _ keyword as placeholder for a parameter. Function (x:Int, y:Int) => x + y can be written as: _ + _, where the _ defines the locations of the two parameters.

1.4.6 Partial Functions

A function that only consists of a pattern match block can be written as a partial function. An example of such a function and the corresponding partial function can be seen in listing 1.8.

```
val bFunc:Function1[BoolLit, Int] =
  (b:Boolean) => b match {
      case BoolLit(true) => 1
      case BoolLit(false) => 0
    }

val bVal:PartialFunction[BoolLit, Int] = {
  case BoolLit(true) => 1
  case BoolLit(false) => 0
}
```

Listing 1.8: Converting a BoolLit into an Int implemented as a function with pattern match (bFunc) and as a partial function (bVal)

bVal is a partial function that maps a BoolLit to an Integer. The type signature PartialFunction[BoolLit, Integer] is defined in a way that the first generic parameter BoolLit is the parameter type of the partial function and the second generic parameter Integer is the result type of the partial function.

If we provide a partial function in an example of this thesis, we will surround the case distinction with curly brackets like in listing 1.9

```
{
  case BoolLit(true) => 1
  case BoolLit(false) => 0
}
```

Listing 1.9: Example of an anonymous partial function

1.5 Viper Intermediate Language

Our rewriter will be used almost exclusively on the AST of the Viper intermediate language throughout this thesis. Therefore, this section introduces and explains the AST nodes that are used in the thesis.

1.5.1 Base Classes

These are base classes that will show up in later examples:

- `Node` is the super class of every Viper AST node.
- `Exp` is the super class of every AST node that represents a Viper expression.
- `Stmt` is the super class of every Viper AST node that represents a Viper statement. A statement is always included inside a `Seqn` node, which represents a list of statements. Therefore, the parent of a `Stmt` is always a `Seqn` and its siblings are the other statements in the `Seqn`. This knowledge will prove useful in Section 3.4.3

1.5.2 Methods

Methods in Viper are functions that have a `Seqn` as body. A method declaration is shown in listing 1.10.

```
case class Method(name:String, args:List[LocalVarDecl],
  rets:List[LocalVarDecl], pres:List[Exp], posts:List[Exp],
  body:Seqn) extends Stmt
```

Listing 1.10: Class declaration of the method AST node

The first parameter `name` is the name of the method.
The second parameter `args` denotes the argument list of the method. They are of type `LocalVarDecl` because parameters behave like method local variables and every declaration of said variables is captures as an AST node of type `LocalVarDecl`.
`rets` contains a list of return variables. In Viper every method can return

multiple values by writing the results into the specifically declared return variables. `pres` and `posts` denote the pre- and postconditions of a method. `body` forms the method body and is a `Seqn` node.

1.5.3 Quantified Expressions

Viper allows quantifiers as expressions. The two supported quantifiers are *forall* and *exists*. We are interested in the common superclass of both quantifiers, since it is important in later examples. Abstract class `QuantifiedExp` is the common supertype of both quantifiers. Listing 1.11 shows the declaration of the class.

```
trait QuantifiedExp extends Exp {
  def qvars: Seq[LocalVarDecl]
  def exp: Exp
}
```

Listing 1.11: Declaration of trait `QuantifiedExp`. It is the super class of `Forall` and `Exists`

`qvars` returns every variable that gets quantified by this expression and `exp` returns the body of the quantifier. Both methods are abstract and are overridden by `Forall` and `Exists`.

1.5.4 If-Then-Else

Viper supports an if-then-else expression (conditional). It is comparable to the ternary operator from other programming languages. The class is shown in Listing 1.12.

```
case class CondExp(cond: Exp, thn: Exp, els: Exp)
  extends Exp
```

Listing 1.12: Class declaration of conditionals

The `CondExp` evaluates to the value of `thn` if `cond` evaluates to true, otherwise `CondExp` evaluates to `els`. If the condition should be non-deterministic, one can use the `NonDet` node. `NonDet` can take either no argument or a list of variables on which the non-deterministic choice depends.

1.5.5 Inhale-Exhale Expressions

The class declaration of inhale-exhale expressions is provided in listing 1.13.

```
case class InhaleExhaleExp(in: Exp, ex: Exp)
  extends Exp
```

Listing 1.13: Class declaration of inhale-exhale expressions

An inhale-exhale expression combines two expressions into one. If an `In-`

`haleExhaleExp` node occurs in *inhaling* position it takes the value of expression `in`. If it occurs in *exhaling* position it takes the value of expression `ex`.

Information in inhaling position can be assumed by the verifier. When proving a method, the verifier assumes the precondition. The information inside the precondition is in this case in inhaling position.

Information in exhaling position has to be proven by the verifier. When proving a method, the verifier has to prove the postcondition given the precondition and the method body, which means that the expressions of a postcondition are in this case in exhaling position.

For more details about inhale-exhale expressions take a look at the Viper paper [6].

1.6 Domain Specific Language

A domain specific language (DSL) is a language specialized to a particular application domain. In this case the application domain is matching on trees. There are two kinds of DSLs we had to chose from.

1. Embedded DSL: Build a language inside the host language (in our case Scala) by using language features of the host language.

   + Advantages come from being inside the host language: access to the full expressiveness of the host language, execution for "free" and IDE support

   − Disadvantage is less flexibility in language design

2. External DSL: Build an own language with parser and interpreter.

   + The main advantage is more flexibility in language design compared to an embedded DSL

   − Disadvantage is less flexibility in language design

   - Disadvantages are no type support and more work because a separate parser and interpreter has to be written

We decided to implement an embedded DSL because expressiveness was more important than design freedom. And in addition to that, Scala's flexible syntax facilitates DSL development. The goals we have for our DSL are listed in Section 1.3.3

Chapter 2

Motivation

This chapter explains the background and the starting situation of the thesis. It introduces a running example that will guide through the rest of this thesis and it backs up the goals we set for this thesis in section 1.3 with arguments and examples.

In this chapter you will encounter Viper code. Viper code should be understandable for experienced programmers. If something is unclear, take a look at Dr. Malte Schwerhoff's PhD thesis [10] which includes a full explanation of Viper's grammar.

2.1 Current Use Cases

Before starting a project it is always useful to evaluate the use cases. Since the Viper project already includes a rewriter, let us take a look on what it is commonly used for:

- Simplifying expressions: It means calculating or reducing an expression already at compile time. E. g. `x := (1 + 11) / 3` becomes `x := 4` or `b := true || x` becomes `b := true`
- Replacing variables: Replacing a variable with an expression occurs rather often in the Viper project. A concrete example would be function inlining, where the occurrences of formal parameters (type `LocalVarDecl`) are replaced with actual parameters (type `Exp`) inside the body of the inlined function. Consider `function inc(x): Int { x + 1 }`. Line `i := inc(i)` would become `i := i+1` through inlining.
- De-sugaring expressions: In this case, de-sugaring means translating a higher level language feature into a lower level language feature to avoid dealing with the high level feature in each backend. As an example directly taken from the Viper language, consider the following:

Viper provides an "is element of" functionality (`in`) for ranged sequences. A ranged sequence is a sequence that includes ascending integers. The `in` method for sequences checks if an element is inside a sequence and returns `true` or `false` depending on whether or not the element is contained in the sequence.
An "is element of" check looks like this: `x in [5..10)`. This check can be rewritten into `x >= 5 && x < 10`.

- Collecting information: The rewriter does not necessarily have to rewrite the AST. It can also be used as a visitor that executes a function with side-effects on every AST node. An example for this usage would be to find out which identifier names are already taken in order to create a fresh collision free variable name.

The new rewriter has to support all the current use cases at minimum and may not require more specification effort than the existing rewriter because otherwise nobody would want to use it. This is captured with Goal 1.

2.2 Rewriting in Scala

This section explains what the standard requirements and features for a rewriter are if it is implemented in Scala. The new and the old rewriter will use the techniques explained in this section.

We introduce basic rewriting with a simple example. The example of choice is evaluating the result of an addition at compile time (simplification). Figure 2.1 shows how this example looks on the AST level.

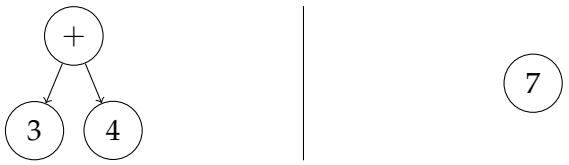

Figure 2.1: Precalculating an addition. On the left side is the original AST and on the right side is the simplified version

The next step is to encode the actual transformation. Pattern matching is the canonical choice for defining a mapping from one AST node to another. If we want to pass such a pattern match around as a function, partial functions (see section 1.4.6) come into play. These partial functions take an AST node as input and produce a new AST node.

```
{
  case And(IntLit(i1), IntLit(i2)) => IntLit(i1 + i2)
}
```

Listing 2.1: Evaluating addition

Listing 2.1 shows how the partial function looks for our example. A rewriting framework can take this partial function and apply it to every node in the AST. If the partial function matches, the node will be replaced with the result of the partial function. If the partial function does not match, nothing will change.

The next question is whether this transformation is to be applied to every node simultaneously, or if it should follow a certain order.

2.2.1 Traversion

The two canonical traversion modes on ASTs are top-down (also known as pre-order) and bottom-up (also knows as post-order). Figure 2.2 shows that the selection of the traversion mode can make a significant difference.

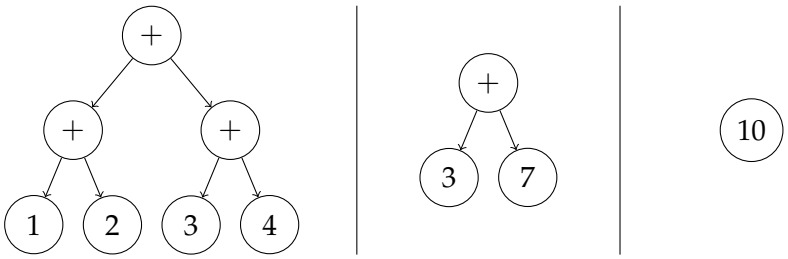

Figure 2.2: Addition of four numbers (left). Partial function 2.1 applied top down (middle) vs. Partial function 2.1 applied bottom up (right).

Selecting Children and Duplication

One has to consider that we are not working with a single type of AST nodes that all have a left and a right child. AST nodes are diverse: `Method` nodes have pre- and post-conditions, a body, arguments and return values as children, whereas a boolean `Not` node only has one expression as a child. Furthermore, not every field of a node should be a child. Fields like name, line number or other meta-information are not important for the AST structure. They also do not extend the `Node` class which defines an AST node in Viper. Therefore, we do not want to consider them as children.
Consequently, one has to write a function that defines where the recursion continues on every AST node. Listing 2.2 shows a part of this function:

```
def recurse(parent: Node): Node = parent match {
...
  case Method(name, args, rets, pres, posts, body) =>
    Method(name, args map go, rets map go,
      pres map go, posts map go , go(body))
  case Not(exp) =>
    Not(go(exp))
  case Add(l, r) =>
    Add(go(l), go(r))
...
```

Listing 2.2: Function that recurses through the AST nodes. Method `go(n: Node): Node` applies the current transformation to a node and returns the transformed node

A problem of this approach is that it is not flexible. If the children of an AST node that should be visited change, a new recurse function has to be provided. But since the recurse function is coded into the rewriter, a change of the `recurse` function implies a change in the implementation of the rewriter.

Another problem with this `rec` method is that every node in the AST is copied during the transformation. This means that node sharing (see Section 2.6) is eliminated in the whole AST as soon as a transformation is applied. Furthermore, every node is duplicated even if it and its children are not involved in any transformation, which is inefficient.

The new rewriter should decouple children selection and duplication and treat them as separate concepts to allow the optimization of both tasks independently. This was stated in goal 3.

2.3 Existing Rewriter

The Viper project already has a working rewriting framework implemented. This will be the baseline where we want to improve from. The following sections explain how this rewriter works and what we want to improve.

```
transform(
  pre: PartialFuncition[Node, Node],
  rec: Function1[Node, Boolean],
  post: PartialFunction[Node, Node]) : Node
```

Listing 2.3: Interface of the existing rewriter

Listing 2.3 shows the interface of the existing rewriter.

- `pre` is a partial function that takes a Viper `Node` as an argument and returns a new `Node`. This function is applied in a pre-order fashion
- `rec` is a function from `Node` to `Boolean` that determines if the recursion continues or stops at a certain node.

- `post` is a partial function with the same properties as `pre`. The difference is that it is used to transform nodes post-order

Now we have everything to implement the addition simplification example from Listing 2.1:

```
val ast: Node = ...
val result = ast.transform(PartialFunction.empty, _ => true,
  {
    case And(IntLit(i1), IntLit(i2)) => IntLit(i1 + i2)
  })
```

Listing 2.4: Simplifying additions with the existing rewriter

Listing 2.4 shows the whole example for addition simplification. Val `ast` contains an arbitrary AST that will be rewritten. It could, for example, be the AST from Figure 2.2. `result` stores the rewritten AST.
The first parameter is `PartialFunction.empty` because the transformation is performed bottom-up and not top-down. An empty partial function as input is defined to transform nothing.
The second parameter is a function that always returns true regardless of the parameter. This ensures that we recurse into every node of the AST.
The third parameter is the partial function from Listing 2.1 that simplifies additions.

2.4 Running example

This section introduces an example that we will use very often during this thesis because it demonstrates a lot of shortcomings of the old rewriter.

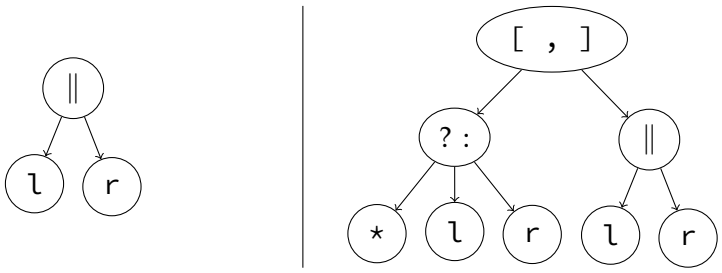

Figure 2.3: Running example: Translating a disjunction into a non-deterministic case-split in inhaling position and the original disjunction in exhaling position

Figure 2.3 shows the basic idea of the transformation. We want to rewrite every disjunction (as indicated on the left) into the expression shown on the right-hand side of the Figure. The right-hand side consists of an `InhaleExhale` expression (`[ , ]`) as the most outside node.
The second parameter of the `InhaleExhale` expression is the matched dis-

junction itself. The first parameter is an if-then-else expression (`?:`). The choice in this if-then-else expression is non-deterministic (`*` operator) and it is a choice between the two operands (`l` and `r`, both boolean expressions) of the disjunction we want to rewrite.

We want this transformation because:
Having a case split in inhaling position instead of a disjunction can be beneficial for automated verifiers, because it can assume `l` at first and try to prove the program and then assume `r` and try to prove the program. In this way the verifier can treat `l` and `r` separately.
Having the disjunction in exhaling position is beneficial because when the verifier has to prove the disjunction, it is easier to prove that `l`, `r` or both hold than it is to find the cases where `l` holds and for the rest of the cases trying to prove that `r` holds.

2.4.1 Caveats

Here we explain the limitations of the existing rewriter when implementing the running example.

Infinite Recursion

The first problem occurs when rewriting the AST in pre-order, which ends in an infinite recursion. This is because the recursion always continues on the rewritten children. The infinite cycle looks as follows: rewrite `l || r` into `[(* ? l : r), (l || r)]`. Then recurse on both children of the `InhaleExhale` expression, namely `* ? l : r` and `l || r`. But `l || r` is the expression we started with and we start all over again.

Quantified Variables

Another problem is that the non-deterministic choice has to depend on every variable that is quantified inside expressions `l` and `r` because the chosen non-deterministic value has to be potentially different for each instance of the quantified variable.
Figure 2.4 shows how the running example looks when variables are bound by a quantifier. `l` and `r` are arbitrary boolean expression that potentially mention the bound variable `x`. The transformation becomes more difficult now because at every transformation step, a list of bound variables has to be present for the non-deterministic choice node.

2.4.2 Transformation

We will build up an implementation of the transformation with the existing rewriter step by step. At first we want to keep it simple and do not worry

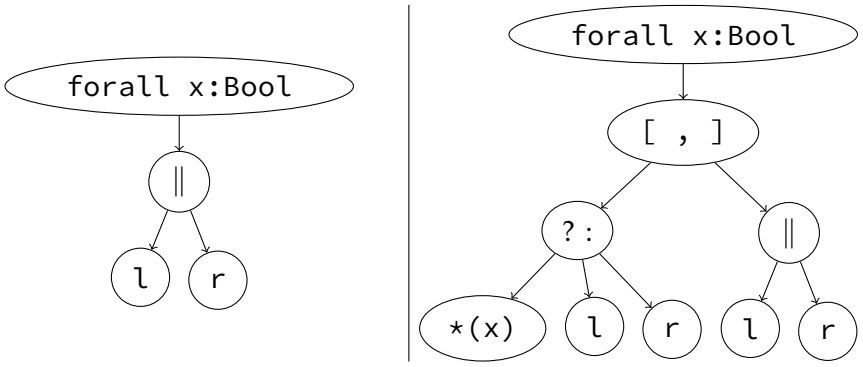

Figure 2.4: Running example: Translating a disjunction into a non-deterministic case-split in inhaling position and the original disjunction in exhaling position

about quantified variables. We only want to perform the rewriting as shown in Figure 2.3. For this we need a partial function that performs the rewriting as explained in section 2.2. Listing 2.5 shows this partial function.

```
{
  case Or(l, r) =>
    val condExp = CondExp(NonDet(), l, r)
    InhaleExhale(condExp, Or(l, r))
}
```

Listing 2.5: Partial function that transforms the AST according to the rule from Figure 2.3

Top-Down The next step is to decide about the recursion order. Since we know about the infinite recursion problem (recall section 2.4.1), a simple top-down traversion is not an option. We either need to write a recursion function that prevents recursing into the exhale part of a newly introduced `InhaleExhale` expression, or traverse bottom-up.

If we want to prevent recursion into certain nodes, we have to implement the `rec` function, seen in Listing 2.3, appropriately. A possible solution would be that the `rec` function captures a `Set` that contains every disjunction that was generated from the rewriting function and if the recursion encounters a disjunction that is contained in this set, the recursion stops.

The new rewriter should be able to do this an a convenient way. This is captured in Goal 4.

Bottom-Up Traversing bottom-up, however is a more convenient solution, since we do not have to write additional code for the `rec` parameter. The drawback in this case is that we transform the exhale parts of the newly introduced `InhaleExhale` expressions because we cannot distinguish between the left and the right child of an `InhaleExhale` expression when traversing

from the leaves to the root.
The result is semantically the same because if `[l, r]` is in exhaling position, it is the equivalent to `r`.

The implementation with the existing rewriter chooses to go bottom-up. In the new rewriting framework we want to be able to do the top-down transformation without suffering from the drawback of additional code explained in the top-down paragraph.

Listing 2.6 shows the transformer utilizing the rewriting function from Listing 2.5 and using bottom-up transformation.

```
val ast: Node = ...
val result = ast.transform(PartialFunction.empty, _ => true,
  {
    case Or(l, r) =>
      val condExp = CondExp(NonDet(),l, r)
      InhaleExhale(condExp, Or(l, r))
   })
```

Listing 2.6: A simplified version of the running example

The final step is adding support for quantified variables, as mentioned in section 2.4.1. For this to be done, we have to make the currently quantified variables available to the `NonDet` node. This is achieved with a `List`, captured by the partial function parameters of the existing rewriter (`pre` and `post`), where we include every quantified variable when going down the AST and remove it when going up again. The implementation is provided in Listing 2.7.

```
val ast: Node = ...
var qVars = List.empty[LocalVarDecl]
val result = ast.transform({
  case q: QuantifiedExp => // (1)
    qVars = qVars ++ q.variables.map(_.localVar)
    q
}, {
  _ => true
}, {
  case q: QuantifiedExp => // (2)
    val qvs = q.variables.map(_.localVar)
    qVars = qVars filterNot qvs.contains
    q
  case Or(l, r) =>
    val condExp = CondExp(NonDet(qVars),l, r)
    InhaleExhale(condExp, Or(l, r))
})
```

Listing 2.7: The complete transformation

The first addition to the transformer is a top-down function `(1)`. This is required because the node that quantifies a variable is an ancestor of the node that uses the variable. Therefore, a function that is applied top-down can collect every quantified variable that will potentially be used somewhere down the AST.
The second addition to the transformer is an extension of the bottom-up transformation `(2)`. We need to remove the quantified variables again when going up, since we are only concerned about the quantified variables from the ancestors of the current node and not the quantified variables we have seen in other branches of the AST.

We want the new rewriter to provide context, such as the quantified variables in example 2.7, with less effort from the user. This motivates goal 2.

2.5 Error Messages

A transformed program may look completely different than the untransformed program. The verifier uses the transformed program for verification and every error message generated by the verifier refers to the transformed program. These error messages could report problems that the user does not see, because the program was changed by the transformation.
Therefore we want to find a way to map the error back into the original program context. Figure 2.5 illustrates this.

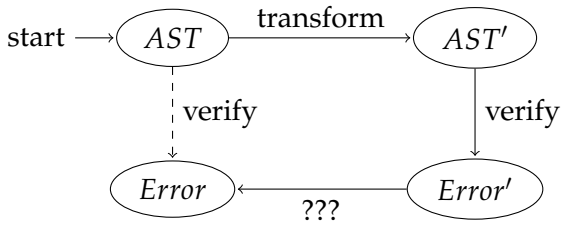

Figure 2.5: Program AST is transformed into program AST'. After verification we get the error $Error'$ for program AST' but what we want is the real error $Error$

An example can be seen in Figure 2.6. The transformed program is obtained by applying a transformation that performs simplification. The corresponding error messages are shown in Figure 2.7.

```
assert 10 <= 5  |  assert false
```

Figure 2.6: Simplification. User writes `10 <= 5`. Verifier sees `false`

```
assert might fail.
10 <= 5 might fail.
```

```
assert might fail.
false might fail.
```

Figure 2.7: Error messages. Left: error message the user wants to see. Right: error message the verifier delivers. User has no idea what `false` means in the context of his program

We want to provide a solution to this problem that works together with the new rewriter. Expectations on the solution of this task are captured in the goals of section 1.3.2.

2.6 Node Sharing

Node sharing is an important topic when working with ASTs. Figure 2.8 shows what we mean when we talk about the sharing of nodes.

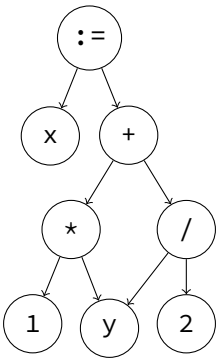

Figure 2.8: Variable y is shared between operators * and /

Node sharing is useful to save memory because e.g. in Figure 2.8 we only need one node for variable y instead of two.

However if we have context sensitive transformations e.g. a transformation that transforms every occurrence of y with x if variable y is inside a multiplication, we cannot keep sharing. The result of this transformation applied on Figure 2.8 is shown in Figure 2.9

These problems with sharing will be considered in this thesis. This topic contributes further motivation for goal 3.

2.7 Deep Matching

Scala pattern matching is used in the existing and the new rewriting framework to match on AST structures. It can be used to match on single nodes

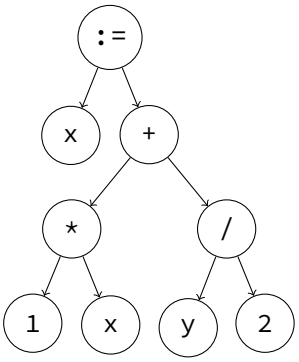

Figure 2.9: Context sensitive transformations can eliminate sharing. Variable y and rewritten y (x) are no longer shared

(`case LocalVar("x") => ...`), or on node structures (`case And(LocalVar("b"), TrueLit) => ...`).

What pattern matching has problem with is matching on structures that are deeper down in the AST (we call it: deep matching). For example if we would want to match on every `And` node that has a variable named `"x"` as descendant in the left operand branch, we have a problem. Pattern matching requires us to specify each node on the path from `And` to `LocalVar("x")` but this is not realistic since the number of AST nodes between `And` and `LocalVar("x")` is arbitrarily high.

To provide a solution for this problem we developed a DSL that is capable of deep matching. This is captured in Goal 10.

2.8 Further goals

2.8.1 General ASTs

One of the potential applications for AST rewriting in the Viper Infrastructure is the expansion of Viper Macros. Macros in Viper behave similar to macros in other languages like C or C++. Macros can be parametrized which means that every parameter occurrence in the macro is replaced with the code that is provided as parameter in the macro call. Listing 2.8 provides an example for a Viper macro:

```
define swap(a, b) {
 var temp := a
 a := b
 b := temp
}
...
var x:Int := 1
var y:Int := 2
swap(x, y)
assert(x == 2 && y == 1)

... expands into ...

var x:Int := 1
var y:Int := 2
var temp := x
x := y
y := temp
assert(x == 2 && y == 1)
```

Listing 2.8: The first part of the listing is the macro definition (macros are defined with the define keyword). The second part shows a macro call. The third part shows how the expanded macros look

Viper macros are expanded during the parsing phase which operates on the Viper Parse AST. In order to utilize the existing rewriter for macro expansion, one has to copy the implementation and replace every Viper AST type with the corresponding Viper Parse AST type.
This motivated us to define goal 5.

2.8.2 Combinable Rewrite Rules

In software development it is a common practice to break one complex problem down into multiple simpler problems. We want provide such a feature for our rewriting library as well.
If a rewriting strategy becomes too complex it might be a good idea to break it down into multiple rewriting strategies and combine them with a combinator. This is captured in goal 6.

Chapter 3

New Rewriter

The following sections explain how the new rewriter works: basic functionality and features that help in achieving the goals are listed in Section 1.3.1.

3.1 Interface

We introduce the new rewriter by presenting the general interface showing the whole set of features. Then we implement the running example and thereby explain the features in detail.

```
val ast:Node = ...
val strategy = StrategyBuilder.Context[Node, NoC](...)
val result = strategy.execute(ast)
```

Listing 3.1: Creating a new rewriter and executing it on AST node `ast`

Listing 3.1 shows how a new rewriter is created from the rewriter factory (`StrategyBuilder`) and executed on AST `ast`. Since the new rewriter provides more features than the existing rewriter we decided to encapsulate the configurations in an object. We call the instances of a rewriter *strategy* because they encapsulate a possible rewriting strategy and they are instances of the `Strategy` class. Listing 3.2 shows how the factory method of the most potent rewriter looks like. This type of rewriter supports every possible feature we provide.

```
def Context[N <: Rewritable, C](
  rewritingFunction: PartialFunction[(N, ContextC[N, C]), N],
  defaultContext: C,
  updateFunction: PartialFunction[(N, C), C],
  traversionMode: Traverse)
```

Listing 3.2: Factory method for the new rewriter

Here we give a small explanation for all the unknown names in Listing 3.2, the following sections will explain every feature in detail:

- `Rewritable`: The rewritable trait has to be implemented by every AST node that should be considered for rewriting. It encapsulates functionality such as children access and node duplication.
- `N`: This type parameter specifies the type of each AST node. It has to implement the `Rewritable` trait at least. This gives the user the possibility to specify a type that is more specific than the `Rewritable` trait to work with inside the `rewritingFunction` and the `updateFunction`.
- `C`: This type parameter specifies the type of the user defined context. We will use a `NoC` dummy parameter in examples to indicate that we do not use user defined context in this example. In Section 3.5 we will see different configurations that allow us to omit the `NoC` type information.
- `ContextC`: This type represents the contextual information that is passed to the rewriting function (we call it: rewriting context). An object of this type encapsulates information about the ancestors, siblings and the appropriate user defined context for a node. The examples in this chapter will bind the parameter of type `ContextC` with variable name `ctxt`.
- `rewritingFunction`: This function is similar to the rewriting function of the existing rewriter. It maps AST node to AST node. Every node in the AST will be given to this function as parameter and the result will be taken as the new AST node. In addition to that, `rewritingFunction` takes a second argument. This is of type `ContextC` and it provides contextual information that can be used in the creation of the new node.
- `defaultContext`: The default user defined context value
- `updateFunction`: The update function defines how the user defined context is built up. Context is gathered from a node by applying this function to a node and the current context to get the new context. If the partial function is not defined, the old context is taken as default instead.
- `traversionMode`: `Traverse` is the type of an enumeration that specifies traversion modes such as `TopDown` and `BottomUp`. If this parameter is not specified `TopDown` is taken as default value.

This looks much more heavy weight than the existing rewriter, especially for smaller tasks. But we also provide rewriter factories that require less configuration effort. They are provided in Section 3.5.

Goal 5 is already achieved by the design of the new rewriter. We implemented no feature that restricts us to one specific AST. The only requirement is that the AST implements the `Rewritable` trait.

3.2 Basics

We illustrate the new rewriter by implementing the running example from Section 2.4. At first we explain the basics.

The contents of this section allow us to achieve **Goal 1**. This means that we can implement the standard use cases of an AST rewriter.

3.2.1 Rewriting Function

The core of the new rewriter is the `rewritingFunction` that maps from AST node to AST node.

```
{
  case (Or(l, r), ctxt) =>
    val condExp = CondExp(NonDet(),l, r)
    InhaleExhaleExp(condExp, Or(l, r))
}
```

Listing 3.3: Transforming Disjunctions into an `InhaleExhaleExp` with a case-split and the disjunction again

Listing 3.3 shows the partial function we use for rewriting. It is exactly the same as partial function from Listing 2.5. The `ctxt` parameter is not in use currently. As soon as we add more complexity to the example we will see the purpose of the rewriting context `ctxt`.

3.2.2 Traversion

We provide the same traversion orders for the new rewriter as we do for the existing rewriter, namely top-down and bottom-up.
They are represented through an enumeration called `Traverse` that contains every transformation mode that the new rewriter supports. This is currently `TopDown`, `BottomUp` and `Innermost` but can easily be extended in the future.

Innermost is a traversion mode that combines top-down traversion with a bit of recursion control. It behaves the same as top-down, as long as nothing is rewritten. As soon as a node is being rewritten, the recursion stops in this branch. This is useful if, for example, the children of rewritten nodes should not be considered for rewriting.

3.2.3 Rewritable Trait

Every node that extends the `Rewritable` trait is eligible for rewriting and we will call such nodes rewritable nodes. The rewritable trait encapsulates necessary functionality such as children selection and duplication. Listing 3.4 shows the two abstract methods of the `Rewritable` trait.

```
trait Rewritable {
  def getChildren: Seq[Any]

  def duplicate(children: Seq[Any]): Rewritable
}
```

Listing 3.4: Interface of the `Rewritable` trait

The `getChildren` and `duplicate` methods provide the required functionality to rewrite a class. The following two subsections give more details about these functions.

The main reason that we have this trait is to decouple recursion (`getChildren`) from the creation of new AST nodes (`duplicate`). By providing this abstraction we have, for example, the possibility to provide a default implementation for `getChildren` and implement `duplication` ourselves. Therefore the `Rewritable` trait is our way to achieve **Goal 3**.

Get Children

The ideal case would be that we can provide the `getChildren` method already in the `Rewritable` interface. This is possible if the rewritable node is a case class:
At first we need to get access to the fields of a rewritable node. As we saw in Section 1.4.2, case classes implement the `Product` trait, which means that we can get a list of fields directly.
If a rewritable node is not a case class, one can implement a field selector via reflection, or the AST node provides a function that returns its children. In the worst case we are at least as good as the existing rewriter.

After we have access to the fields, we need to differentiate between the fields that should belong to the AST and the fields that are properties of a node. To specify that a field belongs to the AST we have the `Rewritable` trait. If a node is rewritable, it is considered an AST node. In case of Viper, class `Node` implements the `Rewritable` trait.
Recall the children selector function from the existing rewriter (Listing 2.2). If we look at, e.g. the `Method` node we see that the fields of this node are:

- `name`: of type `String`. This field does not implement `Rewritable` and is not considered a child

- `args`: of type `List[LocalVarDecl]`. Since `LocalVarDecl` is a subtype of `Node`, it is also a list of `Rewritable`
- `rets`: of type `List[LocalVarDecl]`. Like `args`, a list of `Rewritable` objects
- `pres`: of type `List[Exp]`. Since `Exp` is a subtype of `Node` it is a list of type `Rewritable`
- `posts`: of type `List[Exp]`. Like `pres`, a list of `Rewritable` objects
- `body`: of type `Seqn` which is a subtype of `Node`.

To find out the children of a `Method` node, fields should be considered as children of the AST node if they are either of type `Rewritable` or a collection of `Rewritables`.
With these criteria we can support almost every node of the Viper AST by default. This default children selector is implemented in the `Rewritable` trait and every node that does not match the aforementioned criteria can override this function and implement an own selector.

We use the `Any` type for the children of an AST node because it is a common super type of `Rewritable` and `List[Rewritable]`.

Duplication

ASTs in Scala are often immutable. This means that the nodes of an AST do not contain fields that can be reassigned, everything is constant. If one wants to update a node in an immutable AST, the node has to be copied. The consequence of this is that the parent of the copied node has to be copied as well in order to have the copied node as its child and this has to be repeated up to the root.

If the new rewriter is executed and a new node is generated from the rewriting function, we need to update all the ancestors. Therefore, we need to be able to copy a node with children specified by the rewriter. This is where the `duplicate` method comes in. Every node is required to provide a method that takes new children as arguments and returns a copy of itself with the new children.

```
def duplicate(children: List[Any]): Method = {
  children match {
    case Seq(args:List[Exp], rets:List[Exp], pres:List[Exp],
        posts:List[Exp], body:Exp) =>
      Method(this.name, args, rets, pres, posts, body)
    case _ => throw InvalidChildrenException()
  }
}
```

Listing 3.5: Duplicator of a Method node.

Listing 3.5 shows how a duplicator for our `Method` node looks. The `duplicate` function takes new children (`children`) as parameter. If `children` is a sequence of children that are valid for a `Method` node, the node is duplicated with the new children. Note that the `name` of a `Method` does not implement `Rewritable`, therefore it is not included in the `children` list. Nonetheless the `name` has to be preserved through duplication.

The `getChildren` method and the `duplicate` method of a node need to work with the same children list. Otherwise every call of `duplicate` will fail and throw an exception.

Efficient Duplication Blindly duplicating every node, even if it did not change is not always what we want. The consequences are: preserving shared nodes will not be an option and a slow-down of the rewriting process (if duplication is a costly operation).

On the other hand, one might intentionally want to duplicate every node e.g. to ensure that there is no alias on a node in the tree or to ensure that node sharing is eliminated. This makes sense in the context of, for example, a shared mutable AST. If one of the parties that share the AST wants to rewrite it but the others want to stay with the current AST, the AST has to be copied and then transformed and not just transformed.

To address both problems, the user has the option to choose between duplicating every node and duplicating efficiently. Efficiently means that a node is duplicated only if one of the children changed or the node itself changed. Listing 3.6 shows how to enable or disable efficient duplication.

```
val ast:Node = ...
val strategy = StrategyBuilder.Context[Node, NoC](...)
strategy.duplicateEfficiently
strategy.duplicateEverything
```

Listing 3.6: Enabling and disabling efficient duplication

3.2.4 Application

Listing 3.7 implements the same example with the new rewriter as Listing 2.6 implements with the existing rewriter. Note that we chose top-down traversion instead of bottom-up traversion here because the new rewriter includes features that allow us to deal with the infinite recursion problem conveniently.

```
val ast:Node = ...
val strat = StrategyBuilder.Context[Node, NoC]({
  case (Or(l, r), ctxt) =>
    val condExp = CondExp(NonDet(),l, r)
    InhaleExhaleExp(condExp, Or(l, r))
}, Traverse.TopDown)
val result = strat.execute(ast)
```

Listing 3.7: Simplified running example with the new rewriter

3.3 Recursion Selection

Since we want to implement the running example as a top-down transformation, we need a way to deal with the infinite recursion problem.

There are two approaches to prevent recursion into nodes. The first approach is to provide a recursion pattern for every class that prevents recursion into certain children and the second approach is to prevent recursion into certain marked nodes. This allows us to achieve **Goal 4**.

3.3.1 Stop at Child

Stopping at a child means that every class can select which nodes it recurses into. This is useful if one has to prevent recursion on larger scale than just single nodes. This approach can be looked at as overriding the entries of the children select function.

```
val m: Method = ...
val strat = StrategyBuilder.Context[Node, NoC]({ ... })
strat.recurseFunc({
  case ie:InhaleExhaleExp =>
    List(ie.in)
})
strat.execute(m)
```

Listing 3.8: Preventing recursion into the `ex` child of an `InhaleExhaleExp`

Listing 3.8 shows how you can select individual children for recursion. In this case we select the `in` field but not the `ex` field. Function `recurseFunc` is a member function of the `Strategy`. In case no recurse function is specified, all children are considered for recursion.

This way of selecting children is cumbersome with the existing rewriter. One would need to specify a new recursion function that only recurses into the desired children and select that one instead of the `recurse` function shown in Listing 2.2.

3.3.2 Stop at Node

Stopping at certain nodes allows for a fine granularity of recursion control. This is what we need for our running example since we want to prevent recursion on nodes that are created during the rewriting process.

```
val ast:Node = ...
val strat = StrategyBuilder.Context[Node, NoC]({
  case (Implies(left, right), ctxt) =>
    Or(Not(left), ctxt.NoRec(right))
}, Traverse.TopDown)
val result = strat.execute(ast)
```

Listing 3.9: Preventing infinite recursion in the running example

Listing 3.9 shows how one can prevent the `ex` field of a freshly created `InhaleExhaleExp` instance from being recursed into. Method `NoRec` stores the instance directly into a list of forbidden nodes maintained by `Strategy` object `strat` itself. A useful property of the `NoRec` method is that its return value is its parameter. This allows for concise recursion control as we see in Listing 3.9.

This would not have been cumbersome to implement with the existing rewriter as well. The recursion control of the existing rewriter can only stop the recursion after a node was rewritten and it cannot stop the transformation of the current node.

3.4 Rewriting Context

This section explains how we make the AST nodes aware of their surroundings and provide context to the AST transformations.

We provide access to user defined context, ancestors and siblings through the second parameter of the rewriting function (bound by `ctxt` variable). After this section we will have achieved **Goal 2**.

3.4.1 User Defined Context

This section explains everything regarding the user defined context we presented in Section 3.1. User defined context is a part of the contextual information that is provided to a rewriting function through the rewriting context. When we talk about just context in this subsection we always mean user defined context.

Strategies instantiated by the `Context` factory provide user defined context of type `C` to the partial function used for rewriting. User defined context is gathered from the ancestors of the current node.

For context gathering we provide the ability to specify a function that extracts context from a node and combines it with the already gathered context. The function takes an AST node and the gathered context as arguments and returns the updated context.

Listing 3.10 shows how the context gathering function looks for the running example. It is a partial function that matches on `QuantifiedExp` and the already gathered context of type `List[LocalVarDecl]`. Then it returns the new context which is a new list where the quantified variables of the current node `q` are appended to the already gathered context.

```
{
  case (q: QuantifiedExp, c:List[LocalVarDecl]) =>
    c ++ q.variables
}
```

Listing 3.10: User defined context extractor function

The collected context can be used inside the rewriting function by accessing the field `c` of context object `ctxt`. The complete implementation of the running example to demonstrate how user defined context is used is shown in listing 3.11. The default value for our `List` of `LocalVarDecl`s is the empty list (`List.empty`)

```
val ast:Node = ...
val strat = StrategyBuilder.Context[Node, List[LocalVarDecl]]({
  case (Or(l, r), ctxt) =>
    InhaleExhaleExp(CondExp(NonDet(ctxt.c), l, r),
      ctxt.NoRec(Or(l, r)))
}, List.empty, {
  case (q: QuantifiedExp, c) => c ++ q.variables
})
val result = strat.execute(ast)
```

Listing 3.11: Complete implementation of the running example with the new rewriter

Because we decided to let the partial function accumulate the user defined context, the context can be of any type e.g. a tuple with heterogeneous data or even a single integer that counts something.

N.B. The extracted user defined context comes from the ancestors *including* the current node. The user defined context is different for top-down and bottom-up.

- Top-down: Context is always extracted from the rewritten ancestors. The only exception is the current node because it is not yet rewritten. This also holds for innermost.
- Bottom-up: Since the ancestors are not yet rewritten, the context is extracted from the original ancestors.

3.4.2 Ancestors

In some cases it is required to have access to the ancestors without extracting user defined context from them. Therefore, we provide a complete list of every ancestor of a node to the rewriting function.
Ancestor information is provided through the `ctxt` parameter. Listing 3.12 picks up the addition simplification example we had in Section 2.3 again. The difference in this example is that we only perform the transformation if we are inside a `Method`. To check if we are inside a `Method` we can use the `ancestorList` field of the `ctxt` parameter and check if a `Method` object is among the ancestors.

```
StrategyBuilder.Context[Node, NoC]({
  case (a@And(Int(i1), Int(i2)), ctxt) =>
    if(ctxt.ancestorList.exists(_.isInstanceOf[Method]))
      Int(i1 + i2)
    else
      a
})
```

Listing 3.12: Strategy using ancestor information. The if condition checks if the current node is inside a method declaration. If the condition evaluates to `true` the transformation simplifies the addition, otherwise it returns the addition unchanged. The `a@` before the `And` node binds the `And` node to a local variable `a`.

The following fields are available in the context object regarding ancestor access:

- `ancestorList`: This field provides the whole list of ancestors of the current node, including the node itself. Head of the list is the AST root and the last element of the list is the current node.
- `parent`: `parent` provides the current node's direct ancestor. This is the second to last element of `ancestorList`.

3.4.3 Siblings

In addition to the list of ancestors, we provide access to the siblings of a node. Information about siblings is also contained in the `ctxt` object.
A task that shows the benefit of sibling access is assertion folding. This means that consecutive `assert` statements are merged into one `assert` statement and all the boolean expressions inside the `asserts` are conjuncted together.

```
var acc = List.empty[Exp] // accumulator
StrategyBuilder.Context[Node, NoC]({
  case (a: Assert, ctxt) =>
    acc = acc ++ a.exp
    ctxt.next match {
      case Some(Assert(_)) =>
        NoOp
      case _ =>
        val result = Assert(acc.reduceRight(And(_, _)))
        acc = List.empty[Exp]
        result
    }
})
```

Listing 3.13: Folding assertions using sibling information

Listing 3.13 shows the code for the assertion folding example. The example uses the fact that an `Assert` node is a statement and the parent of every statement is a `Seqn`. The children of `Seqn` are its statements and therefore these statements are pairwise siblings. Figure 3.1 demonstrates this.

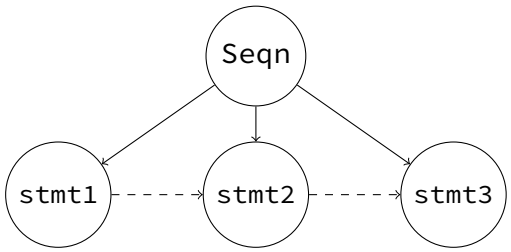

Figure 3.1: This example shows a `Seqn` node with three statements as children. `stmt1`, `stmt2`, `stmt3` stand for arbitrary statements. The dashed arrows denote where field `next` points to from the point of view where the arrows start.

The transformation works as follows: For every `assert` we encounter, we add the current boolean expression to the expression accumulator `acc`. If the sibling which is the direct successor of the assertion we are currently looking at is another assertion, the current assertion becomes a `NoOp` and its boolean expression is added to the accumulator. If no assertion succeeds the current assertion, we conjunct all the accumulated boolean expressions together (`acc.reduceRight(And(_, _))`) and return them in one assertion. Figure 3.2 shows the rewriting step by step. The arrow > indicates the current node.

`acc: List()`	`acc: List(true)`	`acc: List(true, false)`
`>assert(true)` `assert(false)`	`NoOp` `>assert(false)`	`NoOp` `assert(true && false)`

Figure 3.2: The left column shows where we start. The middle column shows that if the first assertion is succeeded by another assertion the condition of the first assertion is put into the accumulator and it is transformed into a `NoOp`. The right column shows that if no assertion follows, the condition of the current assertion is put into the accumulator and a conjunct of the accumulator elements becomes the new assertion condition.

Regarding sibling access, the following fields are available in the context object (`ctxt`):

- `siblings`
- `family`: `siblings` plus the current node
- `predecessors`: Each sibling that comes before the current node in the children list (from `getChildren`) of the `parent` class
- `successors`: Each sibling that comes after the current node in the field declaration list of the `parent` class
- `previous`: Last element of `predecessors`
- `next`: First element of `successors`

If a node has children that are of type collection of `Rewritable`, we consider each element of this sequence as a sibling. Figure 3.3 demonstrates the `next` relation from the example node of Listing 3.14.

```
case class N(c1: Rewritable, c2:List[Rewritable],
  c3:Rewritable, c4: List[Rewritable])
```

Listing 3.14: An example AST node

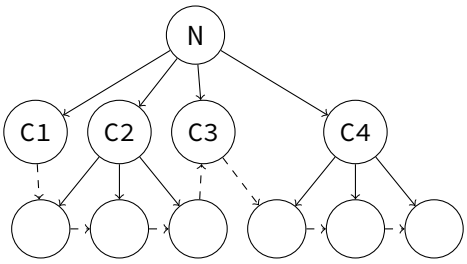

Figure 3.3: This example shows how the sibling relation is interpreted if the children of a node are of type `List[Rewritable]` (example node from Listing 3.14). The dashed arrows denote the value of the `next` field.

3.5 Simpler Configurations

An AST transformation does not always utilize the full set of features that our rewriter provides. We want that no specification overhead is generated for features that are not used. Therefore we provide two other factory methods in addition to `StrategyBuilder.Context`.

3.5.1 Ancestor

If a Strategy wants access to the ancestors or siblings of the current node but does not want to define own user defined context, one can use the `Ancestor` factory method. An example would be Listing 3.13. It uses Sibling information but does not need user defined context.
Listing 3.15 shows what parameters are required for the `Ancestor` factory method.

```
def Ancestor[N <: Rewritable](
  rewritingFunction: PartialFunction[(N,
    ContextA[N]), N],
  traversionMode: Traverse)
```

Listing 3.15: Factory method for transformations that use no user defined context

We can see that in comparison to `Context` (Listing 3.2) `Ancestor` requires less specification.
What changes is the second parameter of the `rewritingFunction`. It is now `ContextA` instead of `ContextC`. `ContextA` has the same functionality as `ContextC` except that is does not include the `c` field for user defined context access.

3.5.2 Slim

Some AST transformations do not require contextual information for rewriting. They only transform one node into another regardless of the surroundings. An example for such a transformation would be the addition simplification example from Listing 2.4.
To increase the convenience for these methods we designed the `Slim` factory method where no second parameter is required in the `rewritingFunction`. Listing 3.16 shows the interface for a `Slim` transformation.

```
def Slim[N <: Rewritable](
  rewritingFunction: PartialFunction[N, N],
  traversionMode: Traverse)
```

Listing 3.16: Factory method for simple node replacement transformations

3.6 Combining Rewriters

We already stated in the introduction that we want to combine rewriting rules in order to allow the user to design AST transformations in a modular manner.

The following two subsections present two approaches to combine rewriting rules. Thereby we achieve **Goal 6**.

3.6.1 Chain Execution

Chaining execution means that two or more strategies are executed one after the other but are treated as one strategy. The following list explains what is possible with our execution chainers. `s1` and `s2` denote `Strategy` objects.

- Sequential chaining (`s1 || s2`): The `||` operator creates a `Strategy` that executes `s1` first. Then takes the resulting AST and execute `s2` on it.
- Repetition (`s1.rep`): The `rep` method of a `Strategy` returns the same `Strategy`, lifted to a repeatable `Strategy`. This means that the execution of `s1.rep` will apply `s1` repeatedly to the AST until the AST does not change anymore. In other words, `*.rep` creates a fixed-point iteration strategy.
  One can provide an integer as an argument to `rep` (e. g. `s1.rep(10)`). Then `s1` will be executed until a fixed-point is reached but a maximum of `10` times.

3.6.2 Interleaving Rules

Interleaving rules means that one can not only execute one strategy after the other but combine them even during the traversal of the AST.

This comes with some restrictions. Only strategies of the same kind can be combined (`Slim[N]` and `Slim[N]`, `Ancestor[N]` and `Ancestor[N]`, `Context[N, C]` and `Context[N, C]`) for reasons of implementation simplicity. The generic parameters of the `Strategy` objects have to be the same. The following paragraphs explains the different rule combinators with the help of a semi-formal notation.

For better understanding we introduce this semi-formal notation: `s1`, `s2` and `s3` denote `Strategy` objects. Let `node` be the node we currently want to rewrite and `sX(nodeY)` be the resulting node from applying `Strategy` `sX` to `nodeY`. `nodeY` $\in$ `dom(sX)` is `true` if the rewriting function `sX` matches on `nodeY` (`nodeY` is in `sX`'s domain) and `false` otherwise.

- Sequential combination (`s1 + s2`): This combines the rewriting functions of `s1` and `s2` together such that on rewriting a node, `s1` is executed and then the result is passed as a parameter to `s2`. That result

is taken as the newly rewritten node. If `s1` does not match, `s2` is executed on the original node.
In our notation this means: `(s1 + s2)(node)` $\leftrightarrow$ `s2(s1(node))`

- Conditional combination (`s1 < s2`): This is very similar to `s1 + s2`. The only difference is that if `s1` does not match on a node, `s2` will not be executed and no rewriting takes place.
`(s1 < s2)(node)` $\leftrightarrow$ `if (node` $\in$ `s1) s2(s1(node)) else node`
- If-Then-Else (`s1 ? s2 : s3`): This construct works as follows: If `s1` does match on a node, `s2` will be executed on the result of `s1`. If `s1` does not match, only `s3` will be executed.
`(s1 ? s2 : s3)(node)` $\leftrightarrow$ `if (node` $\in$ `s1) s2(s1(node)) else s3(node)`

3.7 Summary

In this chapter we introduced the new rewriting framework. The core improvements of the new rewriting framework over the existing framework are:

- Decoupling of duplication and children selection: Section 3.2.3 describes how we separated the functionality of children selection and duplication compared to the old rewriter where one function handled both. The advantage of this abstraction is that we can now treat them as separate concepts and, for example, implement automatic children selection for case classes and efficient duplication independently.
- Recursion selection: Section 3.3 describes the recursion selection feature of the new rewriter. With little specification overhead one can prevent recursion into specific nodes or even nodes that match on a certain pattern.
- Context: The new rewriter provides context to node transformation (see Section 3.4.1). The rewriting function can access ancestors, siblings and even user defined context to assist with rewriting the current node.
- Simpler Configuration: In Section 3.5 we present three different factory methods for creating rewriting strategies. These different factory methods were created to adapt the required specification for strategy creation to a minimum, depending on the amount of features used.
- Combining Rewriters: Section 3.6 presents a way to combine rewriting strategies. This allows the user of the rewriting framework to create complex AST transformation by splitting the transformation into smaller less complex transformations and implementing those as rewriting strategies.

This chapter presented how we can transform a program in AST form into another program. One problem with this approach is that if a program changes, potential errors that might occur during the program may change as well. The next chapter provides a way to deal with this problem.

Chapter 4

Error Handling

We already motivated in section 2.5 that we want to be able to transform an error message back into the untransformed version of the program because the error message should only concern code that was written by the user. The Viper related back-transformations for error messages are not part of the rewriter itself, they are rather an extension of the rewriting framework that was specifically designed for the Viper language.

To understand what has to be done, we first present the structure of error messages in Viper. Then we show how back-transformations work and provide an example.

Meta-Data The term meta-data will occur on various places throughout this chapter. With meta-data we mean data that is contained in the AST nodes but does not define the nodes themselves. In the context of case classes it describes fields that are not in the parameter list and do not have to be mentioned in a pattern match. Meta-data in Viper is: line number, a custom info field and back-transformations that will be introduced in this chapter.

4.1 Viper Error Structure

A Viper error message consists of two parts. An error and a reason for the error. Every error class extends the `AbstractVerificationError` trait and every reason class extends the `ErrorReason` trait. The following two sections give details about errors and reasons.

4.1.1 Errors

An error instance describes what went wrong on a higher level than the error reason. If a program does not verify, the verifier returns one or more

error instances, one for each error that occurred. Examples are:

- `AssertFailed`: Indicates that an assertion might not hold
- `AssignmentFailed`: An assignment might not be possible
- `PreconditionInCallFalse`: A method precondition might not hold
- `PostconditionViolated`: A postcondition might not hold

Every error contains at least the following two fields:

- `offendingNode`: The offending node is the AST node that caused the error. This node is exactly the same instance as the node from the AST we verified, in particular all the meta-data is still present. This will prove useful later in Section 4.2.1.
- `reason`: The `reason` field contains the reason for the error

In this thesis a method `transform` was added to the `AbstractVerificationError` trait that triggers the back-transformation of an error.

4.1.2 Reasons

A reason instance is part of an error instance and describes the error more precisely. Examples for reasons are:

- `AssertionFalse`: States that an assertion evaluates to false. An assertion is a boolean expression that is asserted e.g. by an `Assert` statement, or as part of a method contract.
- `InsufficientPermission`: The Viper language uses a permission model for field access. In order to write a value to a field, one needs to have write permission for that field. If at one point in a program a field is written to without write permission to that field being present, an error including this reason will be the result.
- `DivisionByZero`: The divisor in a division might be zero

A reason also contains the field `offendingNode`. The offending node contains the node instance that is responsible for the reason. The `offendingNode` of a reason is most of the time a subtree of the `offendingNode` of the enclosing error.

4.1.3 Example

Figure 4.1 shows an example of a Viper error message by showing what the user sees. Figure 4.2 show how the object representation of the program and the error message of Figure 4.1 looks. In this example we can also see that the `offendingNode` for an error and a reason is not the same. The error

is more on a statement level and the reason points precisely to the location where the error occurs.

```
var x:Int := 5
assert x < 3
```

```
Assert might fail.
Assertion x < 3 might not hold.
```

Figure 4.1: The left column shows a simple Viper program that contains an error. The right column shows the corresponding error message how it is printed to the user. The first sentence `Assert might fail` is the error and the other sentence is the reason

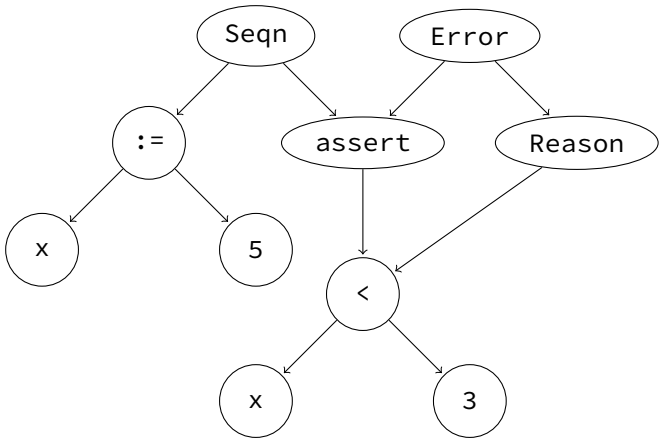

Figure 4.2: This AST shows the program from example 4.1 in AST form and how the error message interacts with the AST. The `offendingNode` field of the `Error` points to the `assert` statement that contains the erroneous code. The `offendingNode` of `Reason` points directly to the location where the error occurs, namely expression `x < 3`.

4.2 Back-Transformations

This section explains how back-transformations work. There are three types of back-transformations: node-, error- and reason back-transformations.

4.2.1 Node Back-Transformation

With node back-transformations we can transform the current node back into the node it was before a transformation was applied.
Listing 4.1 shows an example of how this looks for simplified additions.

```
{
  case a@And(IntLit(i1), IntLit(i2)) => IntLit(i1 + i2)(
    errT = NodeTrafo(a))
}
```

Listing 4.1: Provide a back-transformation (`NodeTrafo`) as meta-data (second parameter list, assigned to parameter `errT`) to the created node. The `a@` before the `And` match is Scala syntax that binds the `And` node to variable `a`.

To add a node back-transformation to a Viper AST node, one can add a node transformation (`NodeTrafo`) to the `errT` field of an AST node. Listing 4.2 shows the interface of class `NodeTrafo`: it takes the original node as the only parameter. This class is used as a wrapper to make it combinable with error- and reason back-transformations. Trait `ErrorTrafo` is the base trait of every error transformation (including error and reason transformations). Field `errT` is of type `ErrorTrafo`.

```
case class NodeTrafo(node: ErrorNode) extends ErrorTrafo
```

Listing 4.2: Interface of class `NodeTrafo`

The following paragraph shows in more detail how a node back-transformation works.

Simplification Example

Let us consider expression `x := 1 / ((2 + 2 + 3) - y)` and its transformation shown in Figure 4.3. If variable `y` might have value 7, we get a Division by 0 error with the offending expression `7 - y`.
To transform this expression back, we recursively apply the back-transformations to the offending expression. In this case root node subtraction (`-`) has no back-transformation appended, node `y` has no back-transformation appended but node `7` has. After the back-transformation, the offending expression is `(2 + 5) - y`. Then we recurse further. Nodes `2` and `+` have no back-transformations appended but node `5` has. After transforming node `5` back we get `(2+2+3) - y` as the offending node, which is exactly what we wanted.

We achieve **Goal 8** because every node stores its back-transformation individually. This means that one can select any node of the AST and transform it back.

4.2.2 Automatic Node Back-Transformation

The rewriting framework introduced in previous chapters provides a method for preserving meta-data across transformations. The idea behind this method

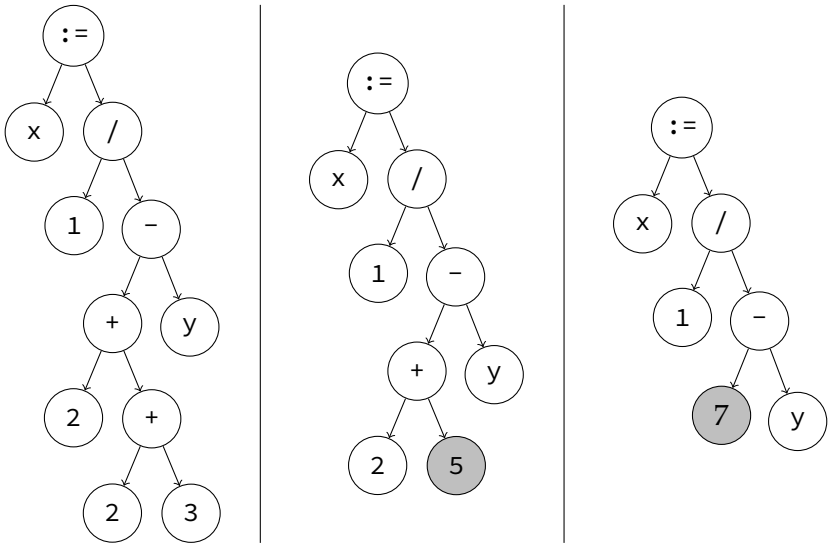

Figure 4.3: Simplifying addition. In the left column is the original AST. In the middle column is the AST where the rule was applied once bottom up and node `5` contains the back-transformation into `2 + 3`. In the right column is the AST where the transformation was completed and node `7` contains the back-transformation into `2+5`, where node `5` is the same instance of node `5` from the middle column.

is that the user is provided with the node before the transformation (`old`) and the node after the transformation (`now`). Then the user can add meta-data from the old node that was not duplicated into the now node. Listing 4.3 shows the interface of this method.

```
def preserveMetaData(old: N, now: N): N = now
```

Listing 4.3: The result of this method is the new AST node. Parameters are: `old:N` the old AST node, `now:N` the new AST node. Returning `now` is the default implementation of this method in the new rewriter.

Method `preserveMetaData` gives the user the option to control node creation. Parameter `old` points to the original AST node and `now` points to the new AST node. By default this function returns the `now` node. The user has the option to override this method. The idea is that meta-data which is not copied during the transformation can be copied over from the `old` node to the `new` node by returning a node that contains everything in method `preserveMetaData`.

We provide a new rewriter (`ViperStrategy`) specifically for the Viper language that overrides method `preserveMetaData` and adds node back-transformations to every transformed AST node. Informally this means duplicating node `now` (since the Viper AST is immutable) and adding line number, info and the node transformation `NodeTrafo(old)`.
This is the only thing required to adapt the general rewriter to a Viper specific rewriter that provides automatic back-transformations.

With this functionality we do not have to provide back-transformation ourselves but the `ViperStrategy` does it for us. **Goal 7** is thereby achieved.

4.2.3 Error Back-Transformations

It is not enough to transform nodes back. We also need to be able to transform errors back. Consider, e.g. the example of method call desugaring. When we desugar a method call we replace it with assertions of the pre- and postconditions of the called method. If an assertion that encodes a precondition check fails, we get an `assertion failed` error instead of a `precondition failed` error.

Specify Back-Transformation To transform error messages, one can add an `ErrorTrafo`, that transforms an `AbstractVerificationError` into another `AbstractVerificationError`, to an AST node the same way a `NodeTrafo` is appended to a node. This means that AST nodes store the back-transformations for the errors they occur in as offending nodes. Listing 4.4 shows the interface of the error back-transformation wrapper. Listing 4.5 shows how the error back-transformation class would look for the method call desugaring example.

```
case class ErrTrafo(error:
  PartialFunction[AbstractVerificationError,
    AbstractVerificationError])
  extends ErrorTrafo
```

Listing 4.4: Interface of class `ErrTrafo` for error back-transformations

```
m.pres.map(pre => Assert(pre)(errT = ErrTrafo({
  case AssertFailed(_, r) => PreconditionInCallFalse(m, r)
}))
```

Listing 4.5: Example of transforming every method precondition of method `m` into an `Assert` statement that checks the precondition. In addition every `Assert` is created with an error back-transformation that transforms the potentially occurring `AssertFailed` error into the according `PreconditionViolated`.

Transform Back Assume we have written a transformer for method call desugaring and the precondition does not hold. Let us further assume that the precondition is an arbitrary boolean expression `bExp`. Then the resulting error is `AssertFailed(Assert(bExp), AssertionFalse(bExp))`. Then we apply the back-transformation of the offending node (as specified in Listing 4.5) to this error message and we get `PreconditionInCallFalse(m, AssertionFalse(bExp))`. The background code for an error back-transformation (without reason back-transformation) can be seen in Listing 4.6.

```
val err = ...
val Terror = error.offendingNode.transformError(err)
```

Listing 4.6: This example shows what happens when an error (`err`) gets transformed back. The offending node of the error stores the back-transformation for potential errors it occurs in as offending node. Therefore it provides a method `transformError` that applies the parameter to a back-transformation that is defined for the parameter (in this case matches `err`) and returns the result. The result is the transformed error `Terror`.

4.2.4 Reason Transformations

A reason back-transformation works in the same way as an error transformation. The offending node stores the reason back-transformation of the reasons it occurs in. Listing 4.7 shows the class interface of the reason back-transformation wrapper.

```
case class ReTrafo(reason:
  PartialFunction[ErrorReason, ErrorReason])
  extends ErrorTrafo
```

Listing 4.7: Interface of class `ReTrafo` for reason back-transformations

The back-transformation of the reason works in the same way as the back-transformation for errors.

4.2.5 Combined

The complete back-transformation of an error (`e`) in Viper happens in the following steps. Listing 4.8 provides the code for these steps:

1. `e.transform` is called by the user of the rewriting framework. The user has full control on the moment when the error is transformed back. This is required because if the error is transformed back too early the backend might get confused because the error does not correspond to the program it got provided as input. Therefore the back-transformation should happen as late as possible.
2. If the offending node of error `e` contains an error back-transformation, we apply it and obtain the new error `ne`.
3. The new error `ne` has two fields: `offendingNode` and `reason`. We first transform the offending node `ne.offendingNode` back according to the specification in section 4.2.1 and get the new offending node `newON`.
4. The second field `ne.reason` is transformed back by applying the reason back-transformations specified in the offending node `ne.reason .offendingNode`. From that we get the reason `newRTemp`. This is temporary because the offending node of `newR` is still not transformed

back. We do this again according to the rules specified in section 4.2.1 and get reason `newReason`.

5. Then we create a new error of the same type as `ne` with offending node `newON` and reason `newReason`.

```
val e = ...
e.transform

def transform = {
  val ne = offendingNode.transformError(this)
  val newON = offendingNode.transformNode

  val newRTemp = ne.reason.offendingNode.transformReason(ne.reason)
  val newRON = ne.reason.offendingNode.transformNode
  val newR = newRTemp.withNode(newRON)

  ne.withNode(newON).withReason(newR)
}
```

Listing 4.8: The step by step explanation from section 4.2.5 in code form. Method `withNode` can be called on an error or a reason and returns the same error/reason with the parameter as `offendingNode`. Method `withReason` can be used to replace the reason field in an error.

This provides a complete error back-transformation of Viper errors and it is what we wanted to achieve with **Goal 9**.

4.3 Summary

In this chapter we provided an extension to the rewriter that allows to provide error messages for transformed programs in a state where no transformation were applied yet.

The requirement to specify the original node for each newly created node may seem like it could introduce a big specification overhead. This is avoided by providing automatic node back-transformations as explained in section 4.2.2.

Error and reason transformations need to be provided by the user since the rewriter cannot find out the connection between the transformed AST nodes and their error messages. Sections 4.2.3 and 4.2.4 explained how such transformations can be specified.
Section 4.2.5 describes the complete back-transformation of a Viper error message.

Chapter 5

Tree Regex DSL

This chapter describes the syntax we motivated in section 1.6.
The goal of the DSL is to make complex matching on nodes simpler and more concise to write. We decided that the DSL should function similar to regular expressions because regular expressions are a well known concept that is used for complex matching. In this chapter we will develop a way to adapt the approach for strings to trees.
The following sections describe how we achieved the goals we set in Section 1.3.3.

5.1 From Trees to Strings

Using a matcher on a string vs using a matcher on a tree is different, because trees are two dimensional structures whereas strings are one dimensional.
Our idea to make string matching work on trees is to consider each so called *"root to leaf"* path of a tree as a string to match on. The "root to leaf" paths of the tree from Figure 5.1 would be: `AB`, `ADAB`, `ADAC`, `ADB` and `ADE`.

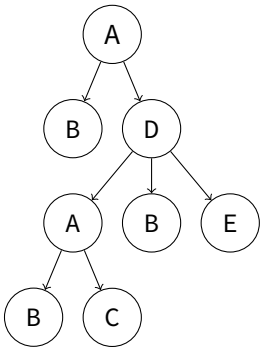

Figure 5.1: A tree with node identifiers

This approach, however, can become very inefficient since a full tree with at least two children per node and depth d has in the best case 2^d "root to leaf" paths we need to check.
Therefore, we use the fact that these "root to leaf" paths share a large amount of nodes among each other and we use the fact that we can apply the matching greedily. Let us explain this with an example.

Simple Consider the simple regex `A(B|C)` (string regex syntax). This regex matches on a path in the tree that starts with a node containing identifier `A`. That node either has to have a node with identifier `B` or a node with identifier `C` as follower.
If we use this regex on the "root to leaf" paths of the tree from Figure 5.1 we get 3 matches. One on `AB`, one on the suffix of `ADAB`, one on the suffix of `ADAC` and no match on `ADE`. This was not efficient because we checked the same nodes `AD` three times.

Improved To make this more efficient we perform the matching directly on the tree. This is done as follows:
We try to start the match on every node in the AST. This is most of the time constant effort since the regex does not match on every node. The nodes that start a matching are all `A` nodes. Figure 5.2 colored the starting nodes in grey.

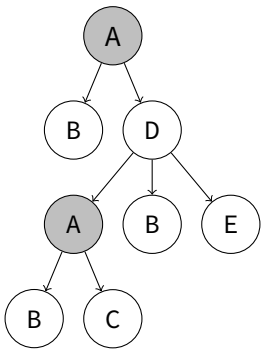

Figure 5.2: Starting nodes for matching `A(B|C)`

The next step is to consume the first match and check if the children match the rest. In this case it means that we reduce `A(B|C)` to `(B|C)` and check the children of all starting nodes `A` for matching. In general, this step is repeated until no match is left, but here we are done already after checking `(B|C)`. Figure 5.3 shows which children match in our example. It gives us the same three matches that we got in the inefficient approach.

This is more efficient because we use the fact that most of the "root to leaf" paths share nodes. Look at e.g. the two "root to leaf" paths `ADAB` and `ADAC`.

The first three nodes are exactly the same. This means that we can find out that the prefix `AD` does not match the regex and that the `A` at the third position might start a match, without treating them as separate "root to leaf" paths already.

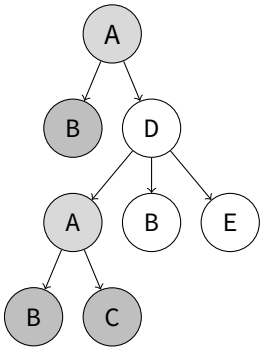

Figure 5.3: Child nodes of As that match `(B|C)`

5.2 Intuition

This section gives an intuition on how our regular expression rewriter works. The details will be provided in the following sections.

Listing 5.1 shows the three core parts of a Regex strategy creation.

```
TRegexBuilder[N, C] &> Matcher |-> Rule[N, C]
```

Listing 5.1: Abstract representation of a regex strategy creation

Matcher The `Matcher` is the core of the Regex strategy. The responsibility of the matcher is to find out which nodes have to be rewritten in the AST and to extract contextual information for these nodes.
We provide matchers and combinators for matchers that are very similar to the ones that exist for strings: `n[And]` for matching on an `And` nodes, `n.Wild` for wildcard matches, `m.?` to indicate that matcher `m` should either match or not, `m.*` to indicate that matcher `m` matches 0 or more times etc.
We also provide matchers that make sense in the context of AST matching such as `c[And](..)` for extracting context from a node or `iC[And](_.left)` to indicate that the matching should only continue on the left child of an `And` node.
A detailed definition of matchers and combinators can be seen in Section 5.3.

Rule A rule is a partial function that defines how nodes are translated, possibly depending on contextual information such as ancestors, siblings or user defined context. These rules have the same signature and behavior as the rewriting functions from chapter 3. They define the rewriting of every node that was marked by the `Matcher` for rewriting.

TRegexBuilder The `TRegexBuilder` creates a rewriting strategy given a `Matcher` and a `Rule`. The `TRegexBuilder` itself specifies the type of the user defined context `C` and the most general node type `N`. A matcher can be added by using operator `&>`. This operator was chosen because of reasons regarding operator precedence in Scala. After a matches is added, one can provide a rewrite rule with `N` as node type and `C` as the user defined context type by using operator `|->`. The result is a new rewriting strategy called "regex strategy". This strategy provides the same functionality as regular rewriting strategies do such as recursion selection, access to ancestors/siblings/user defined context, efficient duplication, etc. A regex strategy can even be part of an execution chain (see Section 3.6.1).
Section 5.5 shows the different factory methods for regex strategies.

Execution A regex strategy is executed in two steps.

1. Generate node-context pairs: The Matcher is applied on the AST and every node that is marked for rewriting is stored with the extracted context as a pair.
2. Rewrite the AST: Every node-context pair gets transformed by a rewriting strategy that uses the rule provided to the TRegexBuilder.

5.3 Matching on an AST

Section 5.1 explained our approach for tree matching in an abstract way with letters as node identifiers. However, we don't need to match on letters but on AST nodes. Since AST nodes have more properties than a letter, e.g. node type, node structures, different children, etc., we want to allow more sophisticated matches.
Section 5.3.7 shows a complete list of all possible matchers we want to support including the required types.

5.3.1 Node Matcher

The simplest possible node matcher is called `n` matcher. An `n` matcher looks like this: `n[T]` where `T` is a generic type that has to inherit from `Rewritable`. In the case of strings the matcher was a letter, e.g. `A`, that matched every occurrence of letter `A`. Here we have, for example, a matcher `n[And]` that matches every `And` instance.

Let us look at the example from Section 5.1. Consider the AST from Figure 5.4, that shows the same tree structure as Figure 5.1, but with AST nodes instead of letters as nodes. These are the replacements: `A` → `And`, `B` → `TrueLit`, `C` → `FalseLit`, `D` → `?:`, `E` → `BVar` where `BVar` is a variable of type `bool`.

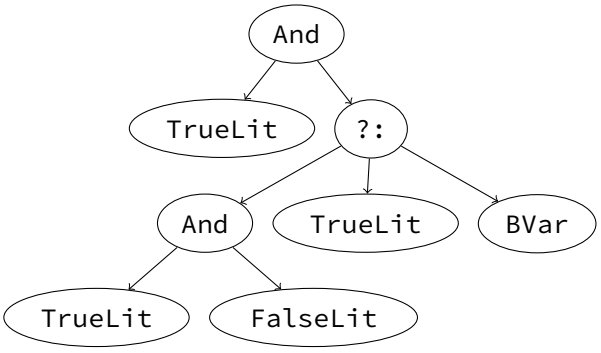

Figure 5.4: AST version of tree 5.1

To perform the matching `A(B|C)` from Figure 5.1 on AST 5.4, we need to match like this: The child of an `And` node is a `TrueLit` or a `FalseLit`. To match on node types we use node matchers: `n[And]`, `n[TrueLit]` and `n[FalseLit]`.
In regular expressions on strings, one writes one matcher after the other (`A` before `(B|C)`) and they are connected implicitly. In the embedded DSL for AST matchers we use operator `>` for connection. Connection in the context of AST matchers means that the left operand is the parent of the right operand. The `|` operator is for choice and it has the same functionality in our DSL than it has in string regexes.
The complete regex looks as follows: `n[Add] > (n[TrueLit] | n[FalseLit])`.

Wildcard Matcher

A wildcard is commonly known to match on everything. In the context of matching AST nodes this means that it matches on every AST node. In our framework one can express a wildcard matcher like this: `n.Wild`.
The wildcard matcher is just a prettier way of writing an `n[Rewritable]` matcher since `Rewritable` has to be the supertype of every AST node.

5.3.2 Child Selector Matcher

The child selector matcher was specifically designed for tree matches and has no equivalent on strings. With a child selector matcher, one can restrict the direction of further matching.

The basis of a child selector match is a node match. We only apply the child selection if the base matcher actually matches. An example instance of a child selector match would be: `iC[And](_.left)`, where `iC` is short for "into child". If this matcher matches on an `And` node, the rest of the pattern will only be applied to the `left` child of the matched `And` node. If the child selector match does not match, nothing happens.

The current implementation of the child selector matcher supports the selection of one child for recursion. If desired one could also implement the selection of a `List` of children. There are no conceptual differences.

5.3.3 Context Matcher

The last matcher that we present is the context matcher. Since our transformations use user defined context (see Section 3.4.1), we want to have a matcher that allows us to collect this context.

Recall our example from Section 3.4.1 where quantified variables were collected as user defined context. Collecting quantified variables looks like this if expressed with the new DSL: `c[QuantifiedExp](_.variables)`. The basis for the context (`c`) matcher is a simple `n` matcher. If the base matcher applies, we use the extractor function (in this case: `_.variables`) on the matched node to collect the context.

5.3.4 Mark for Rewrite

We want to use our regular expressions to rewrite ASTs. But so far we can only find nodes in the AST that match a certain pattern.
Since we want to rewrite AST nodes, we need a way to mark individual nodes for rewriting: every matcher has the option to mark the matched node for rewriting. Every node in a valid path that was matched by a rewrite matcher is considered for rewriting.

Examples for rewrite matchers are: `n.r[TrueLit]`, `iC.r[And](_.left)`, `c.r[QuantifiedExp](_.variables)`.

If we consider the AST of Figure 5.4 again with matcher `n.r[Add] > (n[TrueLit] | n[FalseLit])`, we get the three valid paths again with both `And` nodes marked for rewriting.

5.3.5 Predicate

The presented matchers are still not as flexible as we want them to be: for example matcher `n[IntLit]` matches on every `IntLit`. Matching only on positive or even integers is not a possibility. Therefore, we allowed every matcher to include a predicate that maps every matched node to a boolean value.

If we want to match only on positive integers, we can add a predicate to the matcher: `n[IntLit](_.value >= 0)`. This predicate takes a parameter with the type of the matched node (`IntLit`) and has to return a boolean value (`_.value> 0`).

5.3.6 Combinators

So far, we introduced the two combinators `>` and `|` for our examples. In this subsection we give a complete list of combinators that can be used to combine matchers. Assume that `m1` and `m2` are arbitrary matchers of type `Match` and every combined matcher is also of type `Match`.

- `m1 > m2` : The `>` combinator expresses a child relation. If `m1` matches on a path, `m2` has to match on the direct child path. By direct child path we mean the path that starts directly at the child of the deepest node of the matched path.
- `m1 | m2` : The `|` combinator represents choice. The combined matcher matches if either `m1` matches or `m2` matches.
- `m1.?` : The `?` operator after a match produces a match that applies if `m1` matches 0 or 1 times.
- `m1.*` : The `*` operator works analogous to the Kleene star known from regular expressions on strings. `m1` has to match 0 or more times for the combined matcher to match. It is equivalent to the infinite pattern: `m1.? > m1.? > m1.? > ...`
- `m1.+` : The `+` operator produces a matcher that matches if `m1` matches 1 or more times. The `+` operator is equivalent to: `m1 > m1.*`.
- `m1 ≫ m2` : The `≫` combinator creates a matcher that matches if `m2` matches on a path starting at the descendants of the AST node where `m1` matched. This combinator is equivalent to: `m1 > n.Wild.* > m2`.
- `m1.**` : The intention behind the `**` operator is to match on a node that occurs 0 or more times *consecutively* on a path that is *interrupted* by other nodes. E.g. a path of `And` nodes that is interrupted by an `Or` node would look like this `And-And-And-Or-And`. The purpose of this operator will become clear when we see the example from Listing 5.6 that includes context. This operator is equivalent to: `(m1.* > n.Wild.*).*`

5.3.7 Summary

Regular expressions are a powerful tool to match on strings and we adapted them to use their strengths for matches on ASTs. We address the problem that Scala pattern matching is limited in matching on nodes on a high level and on a deep level of the AST at the same time (explained in Section 2.7). With the regex DSL we achieve **Goal 10**.

In this summary we list all possible matchers:

- Node match:
  - simple: `n[N <: Rewritable]`
  - predicate: `n[N <: Rewritable](p: Function[N,Boolean])`
  - rewrite: `n.r[N <: Rewritable]`
  - rewrite with predicate: `n.r[N <: Rewritable](p: Function[N, Boolean])`
- Child match:
  - simple: `iC[N <: Rewritable](ch: Function[N, Rewritable])`
  - predicate: `iC[N <: Rewritable](ch: Function[N, Rewritable], p: Function[N, Boolean])`
  - rewrite: `iC.r[N <: Rewritable](ch: Function[N, Rewritable])`
  - rewrite with predicate: `iC.r[N <: Rewritable](ch: Function[N, Rewritable], p: Function[N, Boolean])`
- Context match: The context extraction function `con` extracts context of type `Any`. This is because the context matcher does not know which type of context is used for the transformation. It can be used in any type of rewriting. Whether the extracted context has the correct type is checked dynamically at the point where it is actually extracted.
  - simple: `c[N <: Rewritable](con: Function[N, Any])`
  - predicate: `c[N <: Rewritable](con: Function[N, Any], p: Function[N, Boolean])`
  - rewrite: `c.r[N <: Rewritable](con: Function[N, Any])`
  - rewrite with predicate: `c.r[N <: Rewritable](con: Function[N, Any], p: Function[N, Boolean])`

Regex strategies combine the expressiveness of the new rewriter with more powerful matching possibilities. Every strategy generated from the new rewriter can be written as a regex strategy by putting the pattern match of the strategy into a predicate that returns `true` is the node matches and add this predicate to a `n[Rewritable]` matcher.

In general one should choose to use a regex strategy for transformations if complex matching is required.
The advantage of strategies generated from the new rewriter is that they are executed faster because they use standard pattern matching and do not execute the regex matching engine (explained in Section 5.4). Therefore one should use a regular strategy if execution speed matters and Scala pattern matching is sufficient.

5.4 Automatons

To recognize if a matcher matches on a path of AST nodes, we use non-deterministic finite automatons. This is a common approach for regular expressions [2]. Since we built our DSL similar to regular expressions on strings, we can use the same approach with minor modifications.

5.4.1 Specification

Our automaton framework does not only need to recognize a path that matches a regex, it also has to provide information such as: children we select for recursion, a new collected context and nodes that are marked for rewriting. For this task we created our own specification of automatons that got inspired by Khan Process Networks (KPN [4]).

- *States*: A state in our automaton defines which transitions can be taken next. In our automaton design every important information is encapsulated in the transitions. A state is only defined by the transitions that go out of it.
  If a state has no outgoing transitions, we call it a final state. This implies that the automaton is accepting, thereby indicating that the input AST node sequence matched. If no outgoing transition matches the input AST node, the automaton rejects.
- *Matching transitions*: Transitions in an automaton are taken depending on the input. A matching transition models a node match. Therefore, a matching transition is taken if the corresponding matcher matches on the input node. If we write only "transition" in the following sections we always mean matching transitions and not epsilon transitions.
  Furthermore, transitions generate information such as extracted context, children selected for recursion, etc. (similar to actions in KPNs) de-

pending on the matcher being used. This information is then returned as result of the `transition` method (explained in the next paragraph).

- *Epsilon transitions*: An epsilon transition is taken without requiring an input AST node. They make the automaton non-deterministic because an epsilon transition can cause multiple states to be reached on one input.

```
transition(n: Rewritable):(List[State], List[TransitionInfo])
```

Listing 5.2: Interface of the transition method

Every state defines a `transition` method that can be seen in Listing 5.2. This method is called with the AST nodes as parameters. If this method is called with state `s` as target, every transition that can be taken from `s` on input `n` will be taken. The resulting states will be returned as a `List` of `State`. Every information that was generated from the taken transitions is collected and returned in `List[TransitionInfo]`.
For more information about the different `TransitionInfo` classes, have a look at Section 5.4.2.

5.4.2 Construction

This subsection describes how we generate the automaton that recognizes a path in the AST from the DSL specification. For every matcher and for every combinator we define how the corresponding automaton is created.

Matchers

Every matcher yields an automaton that goes into the accepting state for every input AST node that matches and rejects otherwise. This automaton always consists of a single start node, a single end node and a single transition between those two.

The following paragraphs explain how the constructed automatons look. The transition entering from outside denotes the starting node. The state surrounded with a double circle is the final state. Our notation to annotate the transitions is: At first comes the condition which is always a matcher known from Section 5.3. If the matcher matches on the input that was given to the `transition` method (including that the predicate returns `true` if specified), the transition will be taken. Then comes a `/` character that separates the condition from the generated information. This information has to be of type `TransitionInfo`.
A transition with no annotation is an epsilon transition. Details will be explained in the according paragraphs.

Node Match Figure 5.5 shows the automaton created from a node match. The condition of the transition is an `n` matcher. `r` is a `TransitionInfo` that indicates whether the node is marked for rewriting or not.

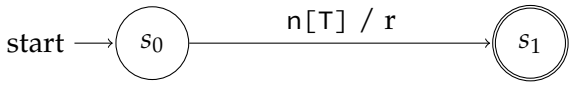

Figure 5.5: A node match automaton

Child Match Figure 5.6 shows the automaton created from a child match. The condition an `iC` matcher. The transition creates two `TransitionInfos`. The first (`r`) is whether or not the matched input is marked for rewriting and the second contains the child `child` that was selected for further recursion.

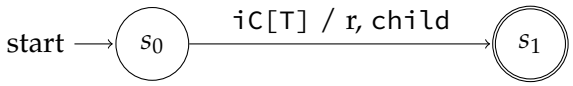

Figure 5.6: A child selection match automaton

Context Match Figure 5.7 shows the automaton created from a context match. The condition is a `c` matcher. The transition creates two `TransitionInfos`: The first (`r`) is whether or not the matched input is marked for rewriting and the second (`udc`) contains the user defined context extracted from the matched node .

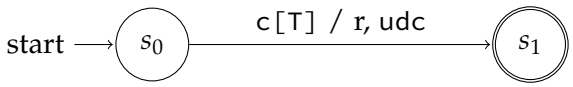

Figure 5.7: A context match automaton

Combinators

Combinators combine two automatons into one automaton or add something to an automaton in case they are unary. The resulting automaton has one starting state and one final state.
We will denote the automatons that are parameters of the combinators as `a0` and `a1`. In the following diagrams we will draw automatons `a0` and `a1` as stars. Every transition that leaves such an automaton is connected to the final state of the automaton and every transition that enters an automaton is connected to the starting state of the automaton.

Direct Child Figure 5.8 shows how combinator > combines two automatons. They simply get executed after each other.

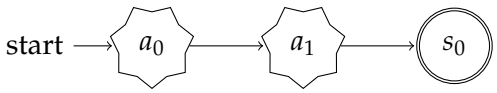

Figure 5.8: Automaton for combinator >

Union Figure 5.9 shows how combinator | combines two automatons. On entering the automaton, the transition goes both ways. This transition then results in two states namely the starting state of a_0 and the starting state of a_1.

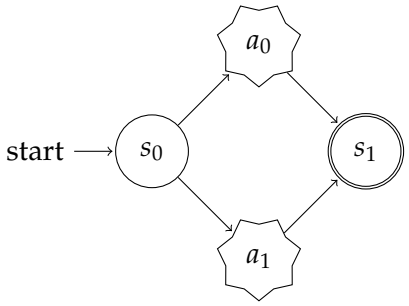

Figure 5.9: Automaton for combinator |

Option Figure 5.10 shows how unary operator ? modifies the input automaton a_0. After entering s_0, both epsilon transitions will be taken and the automaton will be in two states: The starting state of automaton a_0 and state s_1 that skips the execution of a_0.

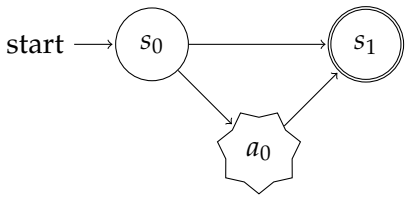

Figure 5.10: Automaton for operator ?

Star Figure 5.11 shows how the automaton modeled after the Kleene Star looks. It can skip execution on a_0 or execute a_0 arbitrarily many times.

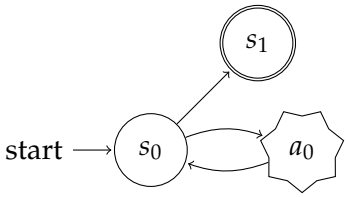

Figure 5.11: Automaton for operator $\star$

Other combinators The other combinators (`+`, `**`, $\gg$) are expressed with the combinators mentioned above. Therefore we do not have to provide an automaton for them.

5.4.3 Match example

Consider the following example: we want to match on and rewrite every implication that has an occurrence of variable `x` somewhere on its right side. The regex for this example would look like this: `iC.r[Implies](_.right)` $\gg$ `n.P[LocalVar](_.name == "x")`.

To start with creating the automaton we need to break down the operations into operations that have an automaton creation rule (in this case $\gg$). The regex resulting from breaking down the $\gg$ operation is: `iC.r[Implies](_.right) > n.Wild.* > n[LocalVar](_.name == "x")`.

Then we can build the automaton according to the rules specified in section 5.4.2. Figure 5.12 shows the whole automaton. The boxes correspond to the automatons generated from `iC.r[Implies](_.right)`, `n.Wild.*` and `n[LocalVar](_.name == "x")`. States that are labeled with e compared to s are states that only have epsilon transitions as outgoing nodes. Those are states in which the automaton will never be in, they are only passed through. We included abbreviations for the transition labels since they would take up too much space otherwise.

iCI: This abbreviates transition label `iC[Implies] / r, _.right`

nW: This abbreviates transition label `n.Wild`. There is no `/` because this transition has only a condition and produces no information.

nV: This abbreviates transition label `n[LocalVar](_.name == "x")`. This transition does not produce information as well.

To show how the matching with an automaton works we need an AST to match on. Figure 5.13 provides an example AST.

Execution

Now we go through the execution of the automaton. We traverse through the AST and provide the nodes as input to the automaton. Since the first

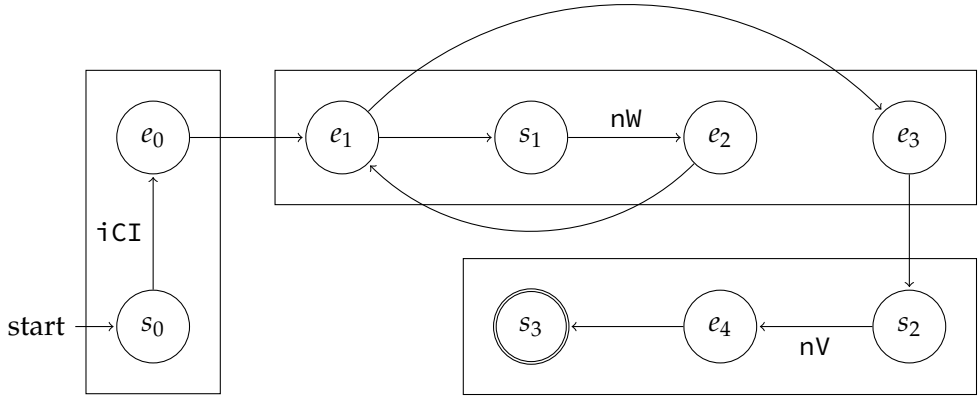

Figure 5.12: The automaton built from regex `iC.r[Implies](_.right) > n.Wild.* > n[LocalVar](_.name == "x")`

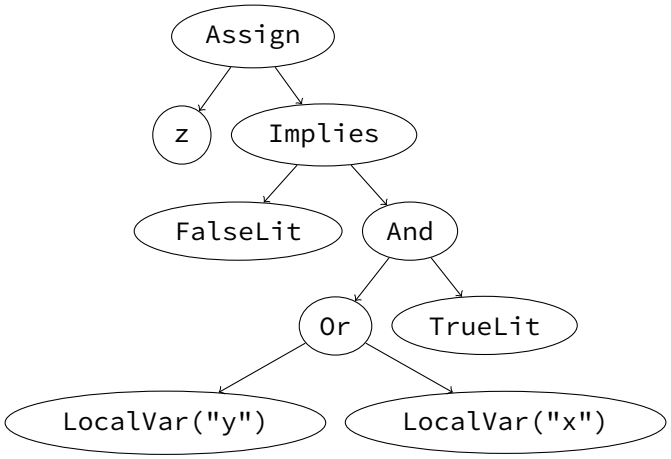

Figure 5.13: An AST we want to match on with the automaton shown in Figure 5.12

transition is only triggered by an `Implies` node, every node rejects on the first input except the single `Implies` node in the AST. The following paragraph explains how the automaton is executed starting from that `Implies` node.

Step-By-Step We execute the automaton on every path, starting at the aforementioned `Implies` node. The notation `Input: Node, {...}` $\rightarrow$ `{...}` denotes that the automaton starts in each state on the left side of the arrow. Then AST node `Node` is provided as input to the automaton (via method `transition`) and after the transition is completed, the automation is in each state on the right side of the arrow.

`Input: Implies, {s0}` $\rightarrow$ `{s1, s2}`: The starting state of the automaton is `s0`. Then we provide the `Implies` node as input to the automaton.

The automaton goes into state `e0` and provides the information that the `Implies` node should be stored for potential rewriting and that only the right child of the `Implies` node is considered as further input. Then the automaton takes every epsilon transition until it reaches a state where it has to consume an input node. If at one point the automaton can take multiple epsilon transitions it takes each of them. After every transition was taken, the automaton ends up in states `s1, s2`.

`Input: And, {s1, s2} → {s1, s2}`: We have to execute the transitions triggered by `And` for each state we are in. State `s1` goes to `e2` since wildcard matches on everything. After every following epsilon transition is chased, we get `s1, s2` again. State `s2` does not have a valid transition for an `And` node therefore we do not get a result state.

`Input: Or, {s1, s2} → {s1, s2}`: We continue with left child `Or`. When we give `Or` as input to the automaton in states `s1, s2` we end up in the same states. `s1` always matches and produces `s1, s2` and `s2` fails on everything except variable `x`.

`Input: TrueLit, {s1, s2} → Fail`: Then we look at right child `TrueLit`. After executing the automaton from `s1, s2` with input `TrueLit` we get `s1, s2` again. Since `TrueLit` was a leaf, the execution stops. We are not in an accepting state and therefore the regex does not match on this path, consequently the `Implies` node is not put into the list of nodes to rewrite.

`Input: LocalVar("y"), {s1, s2} → Fail`: The first child of `Or` is `y`. From `s1, s2` we get to `s1, s2` again. Transition `nV` did not match because the predicate restricts the variable name to be equal to `"x"`. We are in a leaf and at the same time not in an accepting state, the execution fails and the `Implies` node is not put into the list of nodes to rewrite.

`Input: LocalVar("x"), {s1, s2} → {s1, s2, s3}`: The second child of `Or` is variable `x`. Now transition `nV` matches and we go into accepting state `s3`. State `s1` still produces states `s1, s2`. Since a non-deterministic automaton accepts if one of the states in the set is accepting, we found a matching path (`Implies-And-Or-LocalVar("x")`). Node `Implies`, which was marked for rewriting on this path is now added to the final set of nodes to rewrite.

5.5 Application

The purpose of this DSL is to make rewriting more convenient. So far, we are only able to mark nodes for rewriting but cannot perform the rewriting itself. Therefore, we need a way to include a function that defines the node rewriting.
For this task, we decided to integrate the same partial functions we had

for the rewriting strategies from Section 2.3 and Chapter 3. To combine the matching DSL with a rewrite rule, we created an object called `TreeRegexBuilder` that provides different matching strategies similar to the `StrategyBuilder` from Chapter 3.

After this Section it gets clear that regular expressions can be specified concisely for simple tasks because there is no specification overhead generated from the expressiveness i.e. one does not have to specify anything that will not be important for the transformation process. Therefore we achieve **Goal 11**.

5.5.1 Slim Regex Strategy

A simple regex strategy maps from node to node without requiring contextual information similar to a slim strategy from Section 3.5.2. Such a regex strategy is created by using the the `Slim[T]` factory method, where `T` is the most general node type of the AST (`Node` in case of Viper) and `T` has to implement `Rewritable`.
Listing 5.3 shows how the regex from the automaton example (Figure 5.12) looks in Scala. Implications get rewritten into equivalent disjunctions and negations.

```
val t = TreeRegexBuilder.Slim[Node]
val strat = t &> iC.r[Implies](_.right) >>
  n.P[LocalVar](_.name == "x") |-> {
    case Implies(l,r) => Or(Not(l), r)
  }
```

Listing 5.3: Transforming every implication that includes a variable named "x" as a child on the right hand side into an equivalent disjunction and negation

5.5.2 Context

If the partial function should be able to utilize the user defined context that was collected during the matching process, one has to use the `Context[T, C]` factory method, where `T` is again the most general node type of the AST and `T` needs to implement Rewritable. `C` is the type that the rule wants for the collected user defined context. If the matcher collects context of the wrong type (is possible because matcher uses type `Any`), a runtime exception will be thrown. This choice was made because the rule should be statically checkable but the matcher that collects the context should still be decoupled from the rule.

However, simply extracting contexts if a path matches introduces ambiguity problems: Figure 5.14 shows an AST where an ambiguity problem occurs when collecting context with a regular expression strategy from Listing 5.4.

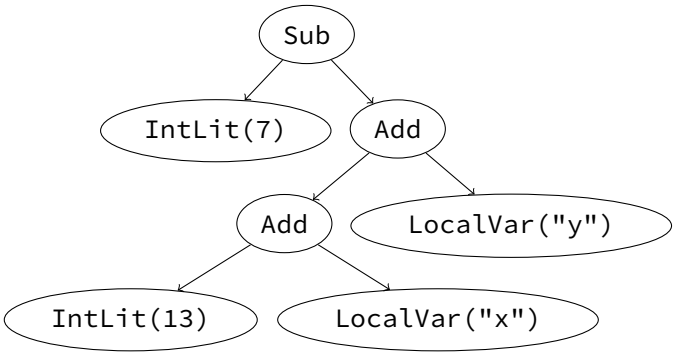

Figure 5.14: AST used to point out ambiguity problems when matching with regex strategy from Listing 5.4

```
val t = TreeRegexBuilder.Context[Node, Int](...)
val strat = t &> c[Add](_ => 1).* >
  n.r[IntLit] |-> {
    case (IntLit(i), ctxt) => IntLit(ctxt.c)
  }
```

Listing 5.4: Regular expression that matches on every `IntLit` and assigns it the number of ancestor `Add` nodes. The three dots indicate that there is functionality missing that is required to handle this transformation.

We encounter the first problem when we match on `IntLit(7)`. Since the context matcher is not required to match once because of the Kleene star, there is the possibility that no context is collected. We address this problem by requiring a `default` parameter for user defined context in the `Context` factory.

The second problem occurs when we match on both `Add` nodes. We extract a `1` from the first node and another `1` from the second `Add` node. But we need to combine them somehow. To solve this problem, we want an `accumulator` method that accumulates the extracted contexts.

The third problem is that we will match on `IntLit(13)` in two ways. The path `Add-Add-IntLit(13)` matches the regex pattern and path `Add-IntLit(13)` matches the regex pattern as well, without the first `Add`. Then we have collected two different contexts for the same node that will result in different transformations. This problem is addressed with the `combinator` method that required the user to specify which context to use if the regex match encounters an ambiguity.

The following itemization explains the general implementation of the three parameters from the `Context` factory method.

- `accumulator:Function2[C,C,C]`: The accumulator function is used to combine the context extracted from a node together with the context

collected already.

- `combinator:Function2[C,C,C]`: The combinator function is used to combine two contexts into one. This is important if the same node is marked for rewriting twice but with different contexts. Then this function selects one context or merges them together into one context.
- `default:C`: The default parameter is used as the starting value for the context and will be present in case no context was collected.

Listing 5.5 provides the complete implementation of the example from Listing 5.4

```
val t = TreeRegexBuilder.Context[Node, Int](
  _ + _, math.max(_, _), 0)
val strat = t &> c[Add](_ => 1).* >
  n.r[IntLit] |-> {
    case (IntLit(i), ctxt) => IntLit(ctxt.c)
  }
```

Listing 5.5: The constructor of the `Context` factory method is now complete. The first parameter is the `accumulator` which is just an integer addition. The second parameter is the `combinator` which returns the bigger number. The third parameter defines the `default` context as 0

The running example from section 2.4 can be written as a tree regex. This was the example where we rewrite disjunctions into inhale-exhale expressions of a non-deterministic choice and the disjunction itself. Listing 5.6 provides this example.

```
val t = TreeRegexBuilder.Context[Node, List[LocalVarDecl]](
  _ ++ _ , (_ ++ _).distinct, List.empty[LocalVarDecl])
val strat = t &> c[QuantifiedExp]( _.variables).** >> n.r[Or]
  |-> {
    case (o:Or, c) =>
      InhaleExhaleExp(CondExp(NonDet(c.c), o.left, o.right),
      c.noRec[Or](o))
  }
```

Listing 5.6: Writing the running example as a tree regex. The context is a List: the accumulator is appending the lists (`_ ++ _`), the combinator merges both lists into one (`(_ ++ _).distinct`) and the default value is the empty list

Listing 5.6 uses the `**` operator to collect the context. Why is the `*` operator not enough? If we use `c[QuantifiedExp]( _.variables).*`, we can only match on consecutive `QuantifiedExp`. In a path `Forall(x)-And-Forall(y)-Or` only variable `x` would be collected and then we match on the `Or` without collecting the other quantified variables.
Therefore, we need to allow arbitrary nodes in between the context matches. This is captured by the following construct: `(c[QuantifiedExp]( _.variables).* > n.Wild.*).*`. We can have an arbitrary amount of interrup-

tions and still match on every quantified variable. This is exactly how `(c[QuantifiedExp]( _.variables).**` is defined.
Note that here the `combinator` function comes into play. The regex with `**` can match in multiple ways on the path `Forall(x)-And-Forall(y)-Or` and not every way includes every quantified variable. If for example `Forall(y)` is matched as a wildcard and not with the `c` matcher then the variable `"y"` will not be collected as context. But since it is guaranteed that the path where both foralls are matched as context objects is among the paths that match and the combinator we wrote merges the lists together, we get `x` and `y` as context.

Since regex strategies created by the `Context` factory use `ContextC` as parameter type (seen in Section 3.1), the rewriting function has access to siblings and ancestors as well.

5.5.3 Ancestors

We also include the option to use ancestor and sibling information without reqiring user defined context. The factory method for this is `Ancestor[T]` with the same rules for `T` than the `Slim` factory.

5.5.4 Behavior on Node Sharing

Recall example 5.5: We count how many `Add` nodes we have as ancestors and assign every integer literal the the number of `Add` nodes in the ancestor list.

If we consider the AST from Listing 5.15 we see that the `IntLit(7)` node is shared among two nodes and the same `IntLit(7)` should be rewritten into `IntLit(1)` and `IntLit(2)`.

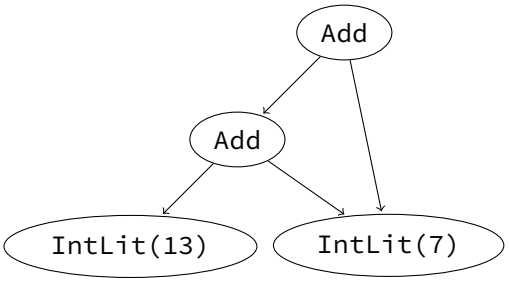

Figure 5.15: AST of expression `13+7+7`. `IntLit(7)` is shared and has both one and two `add` nodes as ancestors.

Our solution to this problem is that we consider the same node with a different ancestor path as different nodes. This means the path `And-Not-IntLit(7)` denotes a different `IntLit(7)` than the path `And-IntLit(7)`.

With this information `IntLit(7)` is marked for rewriting twice as two different nodes and we get the AST of Figure 5.16.

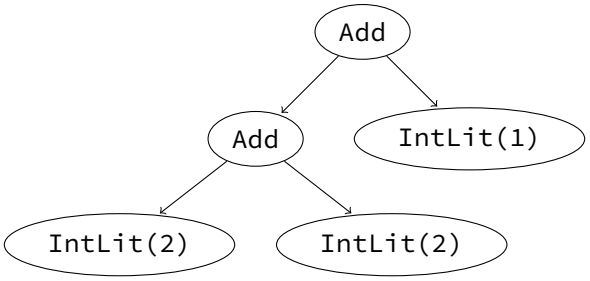

Figure 5.16: Result of applying the transformation from Listing 5.5 to the AST from Figure 5.15.

5.6 Summary

This chapter presented a powerful tool for AST matching. In the first Section (5.1) we utilize concepts of regular expressions on strings for AST matching. We interpret every path on the tree as a string and match on them.

Since AST nodes contain more information than letters in strings we want to have expressive matchers that provide flexibility in node matching. The set of matchers we introduced can be seen in Section 5.3. AST matchers also include functionality that is required specifically for rewriting nodes such as extracting context or marking a node for rewriting.

A regular expression pattern is usually applied by using a non-deterministic automaton. We also implemented an automaton framework to find the matches of a regex pattern on an AST. Our automaton framework is explained in Section 5.4.

The last section before this summary (5.5) shows the different possible regex strategy configurations and how they are applied to ASTs. It also describes how we decided to handle context extraction and the behavior on shared nodes.

Chapter 6

Evaluation

In the evaluation we will again look at the goals we achieved, briefly recap how the solutions work and in some cases suggest improvements left for future work.

To underline these achievements, we provide the implementation of macro expansion and file imports for the Viper intermediate language, which has been based on the new transformation framework

6.1 New Rewriter

Goal 1 *Common transformations that were possible to implement with the existing rewriter should be possible to implement with the new rewriting framework. Furthermore there should be no additional specification overhead generated by the new rewriting framework*

We achieved this goal by removing the existing rewriter completely from the Viper intermediate language implementation and we utilize the new rewriter in every transformation that occurs inside the Viper intermediate language project.
Although the rewriter was replaced, nothing changed for the users since we kept the `transformation` method and only changed its implementation. This means that no additional overhead is generated with the new transformation framework.

Goal 2 *Enhance the convenience when implementing a transformation by making important information about the current node available. This means access to: ancestors, siblings and even special user-defined information.*

The new rewriter provides access to ancestors and siblings when using the `Ancestor` factory method. Additionally, user defined context can be used

via the `Context` factory method. Details on both Strategies can be found in Sections 3.4 and 3.5.

Goal 3 *Children selection and AST node duplication should be treated separately. We want this modularity because it gives us the possibility to optimize and automate children selection and duplication separately.*

The existing rewriter had only one function that defined both children selection and node duplication (see Section 2.2.1).
The new rewriter splits these into two concepts via the `Rewritable` trait. It takes advantage of this split by implementing automatic children selection for case classes and implementing efficient duplication in independent methods (Section 3.2.3).
In Section 2.6 we motivated this goal with node sharing. The existing rewriter eliminated sharing without option. But the new rewriter allows to keep sharing whenever possible because we implemented the duplication function appropriately (see Section 3.2.3).
A possible point for future work would be to utilize reflection for automatic children selection or node duplication. Then the user of the rewriting framework would not have to specify a duplicator or children selector for non case classes anymore.

Goal 4 *The new rewriter should provide a feature to control recursion into AST nodes on the level of nodes and groups of nodes.*

Section 3.3 explains the two possibilities for recursion control. The first option allows to override the children selector and prevents recursion into certain children for each instance of a selected class. This cannot be done with the existing rewriter, one would have to change the core recursion function and thereby change the rewriter itself.
The second option is to prevent recursion into specific AST node objects. This can be done during the transformation with the second parameter of the rewriting function (we called it `ctxt`). Listing 3.9 provides an example.

Goal 5 *The rewriting framework should be able to transform arbitrary ASTs, not only the Viper AST. Furthermore, we want to be able to deal with cycles in order to transform CFGs.*

The new rewriter was designed from the start to handle arbitrary ASTs. Section 6.5 demonstrates the use on an AST that is different from the Viper AST. Section 6.4 shows how cycles in a CFG can be detected and dealt with when using the new rewriter.
The existing rewriter only works on the Viper AST and has to be recreated entirely in order to handle other ASTs. It would require a lot of bookkeeping outside of the declaration of the existing rewriter to detect cycles.

It is possible to detect cycles with the new rewriter, but a small check using the ancestor feature of the new rewriter has to be done manually (see Section 6.4). Integrating such functionality into the new rewriter is a possible way to improve it in the future.

Goal 6 *We want to be able to combine transformations i.e. complex AST transformations can be the combination of many simple transformations.*

The new rewriter allows to combine rewriting strategies in two ways. One way is to chain whole executions of a rewriter, even allowing fix-point iteration. The other way is to interleave the rewriting rules, which offers a lot of flexibility in the design of rewriting rules. A detailed explanation can be found in Section 3.6.
A possible point for future work is to lift the limitations on the combination of different kinds of strategies. Currently only strategies with the same generic types can be combined (details in Section 3.6).

6.2 Error Handling

Goal 7 *The rewriter should provide the ability to transform nodes back into the untransformed state without additional help of the user.*

Section 4.2.2 shows how function `preserveMetaData` is used to automatically generate node back-transformations for the Viper AST. This means that the user has to write a node back-transformation himself only in rare cases.

Goal 8 *The user should be able to select individual nodes, not only the whole AST, and transform them back.*

In section 4.2.1 we show the concept that every node stores the version of itself before the transformation was applied. This means that any node in the AST can be selected and transformed back.

Goal 9 *We want the back transformations to be integrated into the Viper error message framework to map error messages back into the untransformed state (example in 2.5).*

The error message back-transformations were specifically designed for the Viper AST and the implementation directly integrated into the Viper project. Section 4.2.5 shows the back-transformation of a Viper error step-by-step. The transformation of an error message into the untransformed program state can be triggered by calling the `transform` method on the error itself. The user has full control over the point in time where the error is transformed-back.

6.3 Tree Regex DSL

Goal 10 *The DSL should provide a powerful and expressive language to specify transformations. We want the DSL to address the problem of deep matching (explained in Section 2.7).*

Our DSL implementation adapts regular expressions on strings for the usage on ASTs. Regular expressions are a well known tool for dealing with matching problems and we use this power to implement matchers that can match on arbitrary levels of the AST at the same time. The power of our matchers is explained in Section 5.3.
Our DSL is embedded in Scala and therefore we can provide maximum flexibility with, for example, arbitrary Scala functions as predicates inside our matchers. This makes the DSL is expressive and powerful.

Goal 11 *Although the DSL should be designed to be powerful and flexible, standard cases have to be implementable in a simple and concise way.*

Section 5.5 shows how our DSL is used for AST transformations. We provide different factories for regex rewriting strategies such that the specification effort matches the used features. This makes sure that simpler transformations can be encoded with less specification than more complex problems and makes the use of our matching DSL attractive even for smaller problems.

6.4 Handling CFGs

In order to demonstrate that the new rewriter is able to handle cycles in a graph, we will implement a simple example that transforms a simple graph that contains cycles. An actual integration of a cycle detection feature is left as future work.

6.4.1 Example

```
class Node[I](
  var info: I,
  var children: List[Node[I]] = List()
  ) extends Rewritable {

  def addChildren(ch: Node[I]*) = children ++= ch

  override def duplicate(childr: List[Any]): Rewritable = {
    children = childr.collect {  case s:Node[I] => s }
    this
  }
}
```

Listing 6.1: Class declaration of a simple node. The `*` operator at the end of `ch: Node[I]*` defines a variable parameter list similar to `params` in C# or `varargs` in Java.

Listing 6.1 shows the interface of a simple graph node called `Node`. A node only consists of some information `info` and a list of children called `children`.
The `addChildren` function allows to add children after the creation of the node.
The `duplicate` method overrides the `duplicate` method of the `Rewritable` trait. Note that the `duplicate` method does not create a new node in this case, it only updates the current node with the new children. If we would create a new instance when applying the transformation on a node, it would be difficult to close a cycle.

6.4.2 Handling Cycles

```
val a = Node[Int](1)
val b = Node[Int](10)
val c = Node[Int](100)
val d = Node[Int](1000)

a.addChildren(b, c)
b.addChildren(c)
c.addChildren(a, d)
d.addChildren(d)
```

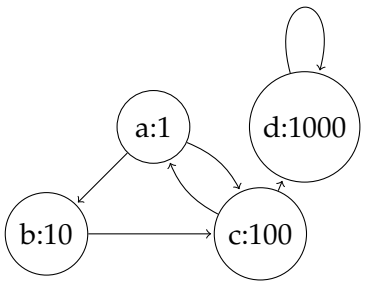

Figure 6.1: On the left column is the code for the creation of the graph represented in the right column.

Consider the graph from Figure 6.1. We want to double the value of every node in the graph without ending in an endless recursion. Listing 6.2 shows how we can do this.

```
var visited = List[Node[Int]]()
val strat = StrategyBuilder.Ancestor[Node[Int]]({
  case (n, ctxt) =>
    if(visited.contains(n)) {
      ctxt.noRec(n)
    }
    else {
      n.info = 2 * n.info
      visited = visited ++ List(n)
      n
    }
})
val res = strat.execute[Node[Int]](a)
```

Listing 6.2: A rewriting strategy that doubles the value of every node in the graph. We use facroty `Ancestor` to get access to the recursion selection functionality (`noRec`).

The transformation works as follows: we match on every node and check if is is contained in the `visited` list. If it is contained, we already transformed this node and would start a cycle. Then we stop the recursion on the current node and return the current node unchanged. Otherwise, we double the `info` field of the current node. After we executed this strategy on node `a`, we get the AST shown in Figure 6.2.
One could also return a new CFG instead of mutating the existing one if the implementation can close the cycle with the duplicated nodes.

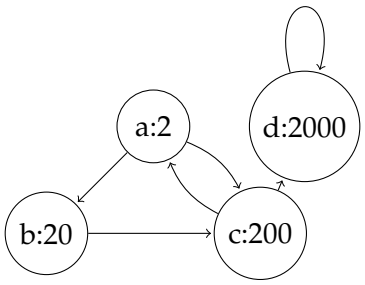

Figure 6.2: The CFG resulting from transformation 6.2

6.5 Viper Imports and Macros

This section will give two examples of rewriter applications that are not on the Viper AST. Both are implemented in the Viper parser which uses its own AST, namely the Viper Parse AST.
The macro expansion and file import features were chosen to be implemented with the new rewriter because they have a recursive structure that can be handled nicely with the new rewriter.

6.5.1 Imports

Viper supports file imports via the `import` keyword. If e.g. `import "library.vpr"` is stated at the beginning of a Viper program, one can use every member (method, function, etc.) that is defined in the program written in file `library.vpr`.

Every member of the main program and of every import gets merged into one program during the parsing phase. This means that our import resolver has to work on the Viper Parse AST. To make the code presented in this section more concise and readable, we assume that a Viper program only consists of imports, macros and methods. Listing 6.3 shows how the simplified Parse AST node for a program looks:

```
case class PProgram(
  imports: List[PImport],
  macros: List[PMacro],
  methods: List[PMethod]) extends PNode
```

Listing 6.3: The simplified class declaration of the Viper parse AST program node

The implementation for parsing methods (`PMethod`) and macros (`PMacro`) is not necessary to know in order to understand the transformation. The `PImport` node wraps a string that contains the file name (field `file`). `PNode` is the super-class of every Parse AST node and implements `Rewritable`.

The basic idea of the import transformation is to perform every import step by step. In each step we take one `PImport` from list `imports`, read the program from the file and convert it into a `PProgram`, then merge its fields `imports`, `macros` and `methods` with the fields of the current program. The imported program get removed from the `imports` list. This is done iteratively until list `imports` is empty.

We only allow to import a file once. Since no file is imported twice, cyclic imports and multiple imports of the same file are handled naturally. Listing 6.4 shows the implementation of this transformation.

```
def importProgram(imp: PImport): PProgram = ...

val imported = Set(f)
val importer = StrategyBuilder.Slim[PProgram]({
  case p: PProgram =>
    val firstImport = p.imports.headOption

    firstImport.match {
      case None => p
      case Some(toImport) =>
        if (imported.add(toImport.file)) {
          val newProg = importProgram(toImport)

          PProgram(
           p.imports.tail ++ newProg.imports,
           p.macros ++ newProg.macros,
           p.methods ++ newProg.methods)
        } else {
          PProgram(p.imports.tail, p.macros, p.methods)
        }
    }
  }).recurseFunc({ case p: PProgram => List() }).repeat
val result = importer.execute(prog)
```

Listing 6.4: The whole rewriter for file imports, `f` is the start file and `prog` is the program parsed from file `f`. Method `importProgram` takes an import statement and parses it into a `PProgram`. `imported.add(..)` is an operation on a Scala set and returns `true` if the added object was not in the set and `false` otherwise.

Explanation The first line contains `val imported`: a set that stores the files we already imported. `f` denotes the file we are currently parsing.

We chose a `Slim` strategy because no context is required during the transformation. We match only on `PPrograms`. The first thing we do is selecting the first `PImport` for importing.
If there are no files to import (`firstImport` is `None`) the transformation does not change anything on program `p`. Otherwise we carry on with the file import.

Then we have to check if this file was already imported. The `add` method of a Scala `Set` returns `true` if the added object was not in the set before and `false` otherwise. If the file was already imported, we just remove the import in the `else` branch.
If this is not the case, we pass the file specified in import to method `importProgram`, which takes a file, parses it, and returns the corresponding `PProgram`. Then we return a new `PProgram` that contains all the fields merged together and does no longer contain the imported import statement.

Since the transformation imports one file at a time we need to iterate until the resulting program does not change anymore, i.e until every import is consumed. This is done using the `repeat` combinator.

If this example would have been implemented with the existing rewriter, one would need to write the code for fix-point iteration manually.

6.5.2 Macros

The other use case for which we use the new rewriter is macro expansion. We show a simplified version of Viper macros to demonstrate how the macro expander works. With respect to macros, this simplified Viper version still exhibits the same challenges as the full version.

Definition

Listing 6.5 shows how a simplified macro node looks like as a Parse AST node.

```
case class PMacro(
  name: String,
  args: List[String],
  body: PNode) extends PStmt
```

Listing 6.5: Class declaration of a macro. The actual Viper node is called `PDefine` instead of `PMacro`.

A macro (class `PMacro`) consists of three fields. The first one is `name` of type `String` that contains the identifier of the macro.
The second is a parameter list (`args`), that stores the formal parameter names. Macro parameters are only names and they do not have types because the formal parameters are replaced directly with the AST nodes that are given as the actual parameters. Since the type-checking phase comes after macro expansion, this does not pose a problem for static type safety.
Field `body` contains the body of the macro which is a statement block or an expression and is of type `PNode`.

Listing 6.6 shows an example of a macro declaration in Viper code. `name` is `"swap"`, args are `List("a", "b")` and `body` holds the statement block between the `{ ... }`.

```
define swap(a, b) {
 var temp := a
 a := b
 b := temp
}

...

var x:Int := 1
var y:Int := 2
swap(x, y)
assert(x == 2 && y == 1)

...

var x:Int := 1
var y:Int := 2
var temp := x
x := y
y := temp
assert(x == 2 && y == 1)
```

Listing 6.6: The first part of the listing is the macro definition (macros are defined with the `define` keyword). The second part shows two macro calls. The third part shows how the expanded macros look

Expanding Macros

When expanding macros we start with a set of macros (we denote it with `macros: List[PMacro]`) and an AST in which the use of those macros should be expanded (we denote it with `toExpand: PNode`).

Helper Functions Listing 6.7 shows the rewriter for macro expansion. It uses four helper functions. Three of them are provided in the beginning of the Listing but one (`replaceParams`) is not because it only purpose is parameter replacement and for a helper method it is too comprehensive to show.
Method `replaceParam` takes three arguments: `body` is the body of the macro, `formalParams` are the parameters of the macro definition and `actualParams` are the parameters of the macro call. The call to `replaceParam` then replaces every occurrence of a formal parameter (in `formalParams`) with the corresponding actual parameter (in `actualParams`) in `body` and returns the resulting AST.

Explanation The rewriting strategy we use for expansion is instantiated with user defined information of type `List[String]`. Since other macros can be called from inside the macro body, we need to watch out for cyclic

calls. Our user-defined information stores the name of every macro we already expanded in the current path.

For historical reasons, method calls are used to represent macro applications in the parse AST. To determine if a `PMethodCall` is a macro call, we wrote the helper function `isMacro` which checks if the name of the called method is the name of a macro. The rewriter matches on every `PMethodCall` *if* it calls a macro.

If we found a macro call, we check for a possible cycle. If we are in a cycle, method `recursionCheck` will throw an exception and the macro expansion stage will fail. Otherwise, we replace the macro call with its body, in which the formal parameters were replaced by the actual parameters.

The context update is rather simple. Every time we encounter a macro call we will add its name to the context.

```
def isMacro(name: String) = macros.exists(_.name == name)

def getMacroByName(name: String): PMacro =
  macros.find(_.name == name) match {
    case Some(mac) => mac
    case None => throw new Exception("Invalid macro name")
  }

def recursionCheck(name: String, ctxt: List[String]) = {
  if (ctxt.macros.contains(name))
    throw new Exception("Recursive macro declaration")
}

def replaceParam(body:PNode, fP:List[String], aP:List[PExp]) = ...

val expander = StrategyBuilder.Context[PNode, List[String]]({
  case (pMacro: PMethodCall, ctxt) if isMacro(pMacro.name) =>
    val name = pMacro.name
    recursionCheck(name, ctxt.c)

    val realMacro = getMacroByName(name)
    val body = realMacro.body

    val formalParams = realMacro.args
    val actualParmas = pMacro.args)
    replaceParams(body, formalParams, actualParams)
}, List.empty[String], {
  case (pMacro: PMethodCall, c) if isMacro(pMacro.name) =>
    val realMacro = getMacroByName(pMacro.name)
    c.macros ++ List(realMacro.name)
})
val expanded = expander.execute(toExpand)
```

Listing 6.7: The complete macro expander. The AST with every macro expanded is stored in `val expanded` in the end.

Hygienic Macros

Listing 6.8 uses the code from Listing 6.6 to swap the value of variables x and y and then swap them back again.

```
var x:Int := 1
var y:Int := 2

swap(x, y)

swap(x, y)

assert(x == 1 && y == 2)

... expand macro ...

var x:Int := 1
var y:Int := 2

var temp := x
x := y
y := temp

var temp := x
x := y
y := temp

assert(x == 1 && y == 2)
```

Listing 6.8: Swapping two times. Since both macro bodies use the same variable name `temp`, we get the error: variable `temp` declared twice.

Macro expansion in Listing 6.8 produces an error because variable `temp` was declared twice. This program is an example of the macro hygiene problem [5].
We solve this problem by creating (partially) hygienic macros. Partially because capturing of variables that are not declared inside a macro is allowed but variables that are declared inside a macro are guaranteed to not collide with any variables declared outside.

To achieve this, we generate a fresh variable name for every variable that was declared inside a macro. With our hygienic macro implementation the example from Listing 6.8 becomes Listing 6.9.

```
var x:Int := 1
var y:Int := 2

swap(x, y)

swap(x, y)

assert(x == 1 && y == 2)

... expand macro ...

var x:Int := 1
var y:Int := 2

var temp := x
x := y
y := temp

var temp$1 := x
x := y
y := temp$1

assert(x == 1 && y == 2)
```

Listing 6.9: Swapping two times with hygienic macro expansion. Fresh variables are created by adding a $ and a counter vale, to avoid collisions with existing variable names.

6.6 Summary

The first three sections of this chapter list all the goals we set for this thesis and give a concise explanation on how we achieved them. It also states potential extensions for the new rewriting framework as future work.

In Section 6.4 we provide a short example of the transformation of a simple circular graph to undermine the claim that our rewriting framework provides the tools to deal with CFG transformations.

Section 6.5 describes the implementation of file imports and macro expansion. Both are important tasks for parsing a Viper program and both tasks are now implemented as rewriting strategies.
One reason behind these transformations was to show that the new rewriting framework can handle arbitrary ASTs and not only the Viper AST.The other reason is that both tasks are ideal applications for AST rewriting and we utilize the new rewriting framework to provide a readable and clean implementation for them. We used this opportunity to deal with the hygiene problem of Viper macros by making macro expansion (partially) hygienic.

Chapter 7

Conclusion

In the course of this thesis we developed a powerful rewriting framework. Chapters 1 and 2 introduce the necessary project background and motivate our goals for this thesis. They are achieved step by step in the following chapters.

Chapter 3 describes a powerful library for AST transformations. We implemented useful features such as recursion control, ancestor and sibling access, user defined information, etc.

Chapter 4 introduces error back-transformations. In this thesis, these are used to express the error message of a transformed program in the context of the original program.

Chapter 5 provides a DSL to help with expressing complex matching patterns. The DSL implements regular expressions for ASTs. Because ASTs are different from the usual application field of regular expressions which are strings, we had to define how a regular expression is applied to a tree. We defined own node matchers and operators. Matching a regular expression on an AST is performed with a non-deterministic finite automaton framework developed in this thesis.
The DSL is very flexible and allows access to the features of the rewriting framework (ancestors, siblings, user defined information, recursion control, etc.).

Chapter 6 comments on the goals we achieved and provides two important use cases of the new rewriter inside the Viper parser.

7.1 Future Work

Although we achieved the goals we set for this thesis there is still room for improvement. The following list suggests tasks to further improve the new rewriting framework.

- Section 3.2.3 introduces the `Rewritable` trait that decouples children selection from duplication in order to allow the automation of those two.
  Reflection could be used to implement a default children selection and/or node duplication for general classes. Then the user of the rewriting framework would not have to specify a duplicator or children selector for non case classes anymore.
- In Section 6.4 we show that the new rewriter is conceptually able to handle CFG transformations. Viper also has a CFG representation, but we did not implement a strategy for it that integrates cycle handling.
- Combination of strategies (see Section 3.6) is currently restricted to combining strategies created from the same factory method and with the same generic types. However, different combinations would be possible in theory e.g. `Context` with `Slim` and the `Slim` strategy ignores the gathered context information.
- The automaton that recognizes whether an AST matches a regex is non-deterministic (see Section 5.4). The current implementation exhibits bad performances if the automaton includes a lot of wildcard matches or Kleene stars. This is a common problems among non-deterministic automatons that recognize regular expressions and not specific to our regex engine [8]. It would be beneficial to investigate how the automaton could be optimized in order to improve performance.

Bibliography

[1] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/xt 0.17. a language and toolset for program transformation. *Science of Computer Programming*, 72(1–2):52 – 70, 2008. Special Issue on Second issue of experimental software and toolkits (EST).

[2] Anne Brüggemann-Klein. *Regular expressions into finite automata*, pages 87–98. Springer Berlin Heidelberg, Berlin, Heidelberg, 1992.

[3] Leo Büttiker. Rewrite engine for staged expressions. Master's thesis, ETH Zurich, Advanced Computing Laboratory, 2013.

[4] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam.

[5] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 151–161, New York, NY, USA, 1986. ACM.

[6] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016.

[7] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-Step Guide, 2Nd Edition*. Artima Incorporation, USA, 2nd edition, 2011.

[8] Thielecke H. Rathnayake, A. Regular expression matching and operational semantics. In: Structural Operational Semantics (SOS 2011). Electronic Proceedings in Theoretical Computer Science (2011).

[9] Scala documentation. http://docs.scala-lang.org/. Accessed: 2017-03-22.

[10] M. Schwerhoff. *Advancing Automated, Permission-Based Program Verification Using Symbolic Execution*. PhD thesis, ETH Zurich, 2016.

[11] Anthony M. Sloane. *Lightweight Language Processing in Kiama*, pages 408–425. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[12] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames. *ACM Trans. Program. Lang. Syst.*, 34(1):2:1–2:58, May 2012.

[13] Tregex. https://nlp.stanford.edu/software/tregex.html. Accessed: 2017-03-24.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

A Framework for Bi-Directional Program Transformations

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Fritsche	Simon

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zurich, 2017-04-07	F Simon

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.