# A Framework for Bi-directional Program Transformations

Student: Simon Fritsche
simonfri@student.ethz.ch

Supervisor: Malte Schwerhoff
malte.schwerhoff@inf.ethz.ch

## I. Introduction and Background

Program transformations are a common technique to improve program properties such as speed and readability. A program transformation takes a program represented by an AST (abstract syntax tree) and applies a function to transform it into another AST. Examples for widely used transformations are: constant folding, common sub-expression elimination and copy propagation. Details about such transformations can be found in [2]

The Chair of Programming Methodology at ETH Zurich wants to use such transformations for the Viper infrastructure [1].
The Viper infrastructure is designed for modular verification of high-level programming languages. Verification is achieved in two steps:

- A front-end encodes a high-level program into the Viper intermediate language
- A verifier verifies the Viper intermediate language (back-end)

## II. Applications

### A. Desugaring

Program transformations can be used for desugaring. This means that we can offer a large set of language features for the front-end, that is then transformed into a smaller set of language features. Therefore, the back-ends only need to deal with a reduced language.

```
var i:Int := 0

while (i < 5)
  invariant i <= 5
{
  i := i + 1
}
```

```
var i:Int := 0

assert i <= 5
if(!(i < 5))
  goto skiploop

label whileloop
  i := i + 1
  assert i <= 5
  if(i < 5)
    goto whileloop

label skiploop
```

Fig. 1. Desugaring a while loop

Figure 1 provides an example and shows, how one would express a loop by using **if** and **goto**.
The assertions on the right side correspond to the invariant of the while loop and represent checks on entry and after the execution of the loop body.

### B. Property checks

Checks that the verifier had to do itself can be encoded explicitly into the code via a transformation, making it easier for the back-end programmers because they don't need to program those checks into each individual back-end anymore.

Examples of such checks are well-definedness checks such as recognizing division by 0 (example in Figure 2) and checks that detect inconsistent states during verification (smoke checks).

```
var i:Int := 0

assert 5 >= 10 / i
```

```
var i:Int := 0

assert i != 0 &&
       5 >= 10 / i
```

Fig. 2. A well-definedness check

### C. Tree manipulation

We want to use the transformation framework also for AST transformations that currently happen in the back-ends of Viper. The complexity of such transformations could range from simple examples such as renaming variables, to more complex examples, such as the following:

$$\forall i : e_1(i) \parallel e_2(j) \rightarrow [*_i \ ? \ e_1(i) \ : \ e_2(j), \ e_1(i) \parallel e_2(j)]$$

Fig. 3.

The left-hand side of the arrow is the expression we want to replace with the right-hand side. There we see a pair of expressions separated by a comma inside square brackets $[exp_1, exp_2]$. This forms a composed expression where $exp_1$ is used inside assumptions and $exp_2$ is used in proof obligations.
The first expression contains a $*$ operation as the condition of a ternary operator. The $*$ denotes a non-deterministic choice, which has to depend on all variables that are quantified in the expressions we chose from. Therefore, the $*$ in Figure 3 depends on $i$ but not on , which is a local variable.

If we want to replace every disjunction with the corresponding right-hand side, we need a way to match on every disjunction we encounter, when traversing the AST. We

also have to know which variables used in disjunctions are quantified and which are not because the $*$ operator needs to depend on the quantified variables. This means that we have to provide a context for the transformation rule in which we store the currently quantified variables.
For such tasks we want an efficient, yet expressive way of specifying transformations. A sketch of the specification language we propose is provided in Figure 4

$$(\forall i)^* \ >> \ e_1(i) \parallel e_2(j) \rightarrow$$
$$[*\,(context)\ ?\ e_1(i)\ :\ e_2(j),\ e_1(i) \parallel e_2(j)]$$

Fig. 4. Sketch of a specification language

Figure 4 illustrates in which direction we want to go with the encoding. The $(\forall i)^*$ describes the $context$ which is then used in the right-hand side of the transformation. It is written in the style of a regular expression and its intuitive semantics is that we collect every quantified variable on the way to the disjunction.
The disjunction $e_1(i) \parallel e_2(j)$ is the expression we match on. This will be transformed with our rule.
On the right side of the arrow we have the expression we want, as described in Figure 3. Note that we can now use the context to get information about the variables that are quantified and provide them as arguments to the $*$ operator.
Further details about the transformation encoding can be found in section IV.

## III. Error Reporting

A transformed program may look completely different from the original program. The back-end uses the transformed program for verification and every error message emitted by the back-end 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 need to find a way to map the error back into the original program context.

A small example can be seen in Figure 5. The transformed program is obtained by applying a transformation that performs constant folding. The corresponding error messages are shown in Figure 6.

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

Fig. 5. Constant folding

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

Fig. 6. Error messages

## IV. Transformation Encoding

Our transformation encoding should on the one hand be simple and easy to use but on the other hand be very expressive such that complex transformations can be encoded as well. Therefore, we decided that our encoding should have two layers.

The first layer is an *embedded DSL* (domain specific language) that allows the user to use arbitrary code of the host language in the definition of the transformations to make complex transformation possible.
The advantage of an embedded DSL over an *external DSL* is that the embedded DSL can use the type system, IDE support and other conveniences that the host language brings.
An external DSL, however, provides more flexibility regarding the syntax as it is not bound to the host language. In our case we value functionality over flexibility and we therefore chose the embedded DSL.

The second layer will provide in particular a more user-friendly syntax with regular expressions on trees for more convenient matching that will then be translated and expressed via the first layer.

## V. Core Goals

During the course of this master's thesis we want to achieve the following goals:

- Collect ideas for the embedded DSL and regular expressions on trees from projects such as tregex/tsurgeon [3] as well as the master's thesis of Leo Büttiker [4]
- Collect potential use-cases
- Develop a mechanism to map error messages back into the original program context
- Design and implement a convenient embedded DSL for AST transformations
- Design and implement the second layer language for a more user-friendly syntax
- Evaluation of the framework by providing a core set of useful transformations ready to use

## VI. Extended Goals

After the core goals have been achieved, a subset of the following extended tasks can be tackled:

- Implement support for the Viper CFG
- Implement support for program transformations on general ASTs or CFGs
- IDE support for applying selected transformations

## References


[1] 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, VMCAI, volume 9583 of LNCS, pages 4162. Springer Verlag, 2016

[2] Aho, Alfred V. and Sethi, Ravi and Ullman, Jeffrey D., Compilers: Principles, Techniques, and Tools. Addison-Wesley Longman Publishing Co. Inc., Boston, MA, USA. 1986, isbn: 0-201-10088-6.

[3] http://nlp.stanford.edu/software/tregex.shtml

[4] Leo Büttiker: Rewrite Engine for Staged Expressions, 2013, Advanced Computing Laboratory ETH Zurich