Toward Abstract Interpretation of Program Transformations

Sven Keidel
Sebastian Erdweg
TU Delft

Abstract

Developers of program transformations often reason about transformations to assert certain properties of the generated code. We propose to apply abstract interpretation to program transformations in order to automate and support such reasoning. In this paper, we present work in progress on the development and application of an abstract interpreter for the program transformation language Stratego. In particular, we present challenges encountered during the development of the abstract Stratego interpreter and how we intend to solve these challenges.

ACM Reference Format:
Sven Keidel and Sebastian Erdweg. 2017. Toward Abstract Interpretation of Program Transformations. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 5 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Program transformations are meta programs that translate code of an input language to code of an output language. Examples of program transformations are compilers, interpreters, refactorings, normalizations, etc. Developers of program transformations often reason about transformations to assert certain properties of the transformations' output:

- Does the generated code of a compiler have the same semantics as the input program?
- Is the generated code of a compiler efficient?
- Does an interpreter always return values for well-typed programs or can it get stuck?
- Does a refactoring always produces well-typed code?
- Does a normalization transform all programs to a normal form?

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

One way of answering these questions is by formal verification with a proof. However, most such proofs are manual and happen after the development of program transformations, meaning they cannot be integrated into the development process.

Another way of reasoning about program transformations is by static analysis, also known as *lightweight verification.* A static analysis inspects the code of a program transformation and can compute properties about how the output of the transformation looks like. The results of the analysis can then be inspected by developers to find counter examples or confirmation for the properties from above. Compared to unit testing, a static analysis requires no input programs for a program transformation since it computes properties that hold for all inputs. Compared to manual verification, a static analysis can be run automatically to continuously provide feedback within an IDE.

In this work, we propose the static analysis of program transformations by abstract interpretation [Cousot and Cousot 1977] and describe the challenges imposed by this approach. In particular, we present early results about our abstract interpreter for the Stratego program transformation language [Visser et al. 1998], which we are currently developing. The abstract interpreter approximates the output of program transformations without information about the input program. The results of the analysis can be inspected to get feedback for a given program transformation. This can be used to find counter examples to the properties mentioned above or for debugging a program transformation during the development process. Based on our early experience with abstract interpretation for Stratego, we describe the challenges we encountered and how we intend to solve them.

2 Abstract Interpretation of Stratego

In this section, we demonstrate our abstract interpreter for the Stratego [Visser et al. 1998] program transformation language by example. As example program transformation, we consider an evaluator for arithmetic expressions, which we expand throughout the paper and show what the effects of each change on the results of the analysis are. Note that only the last version of our "evaluator" will actually normalize expressions to values, whereas the other versions produce intermediate evaluation terms.

Stratego transformation are commonly defined as a set of rewrite rules. A rewrite rule matches a pattern against an

input term and produces an output term by instantiating a template. We start our exploration by defining a rewrite rule for multiplication by zero.

```
eval: Mul(Zero(),_) -> Zero()
```

When running this evaluator with the concrete Stratego semantics, we can observe one of two possible outcomes: If the input term matches the pattern `Mul(Zero(), _)`, then the output is `Zero`, but if the input term does not match the pattern, then the whole transformation fails. To be sound, an abstract interpreter must approximate all possible outcomes of the concrete semantics for an unknown/undefined input term. Indeed, for this simple example, our abstract Stratego interpreter predicts the same results the concrete semantics computes:

Concrete Stratego	Abstract Stratego
`Fail, Zero()`	`Fail, Zero()`

To handle more arithmetic expressions, we add further rewrite rules to the evaluator:

```
eval: Mul(Zero(), _) -> Zero()
eval: Add(Zero(), n) -> n
```

If the first rewrite rule fails, the concrete interpreter tries to apply the second rule and only emits failure if that one fails as well. The second rewrite rule copies the second operand of `Add` unchanged to the output. Consequently, the concrete interpreter can produce infinitely many different output terms depending on the input term. The abstract interpreter must predict the outputs of the transformation without knowing about the input term. On the one hand the abstract interpreter cannot produce an infinite set of alternative output terms, on the other hand soundness requires the abstract interpreter to predict all possible outputs. The solution is to approximate the infinite set of output terms using a finite representation. Specifically, our analysis uses the nonterminal symbol `Exp` to summarize all possible terms copied by the second rewrite rule:

Concrete Stratego	Abstract Stratego
`Fail, Zero(), Succ(Zero()),` `Mul(Add(...), Zero()), ...`	`Fail, Zero(), $Exp`

Note that it would have been sound to drop `Zero()` from the output of the abstract interpreter because `$Exp` subsumes it. However, we choose to keep track of subsumed results because they witness specific terms that were encountered during the interpretation. For example, in an evaluator for arithmetic expressions, tracking `Zero()` makes sense and can provide added precision.

As we add more rewrite rules to the evaluator, we obtain a larger set of possible outputs:

```
eval: Zero -> Zero()
eval: Mul(Zero(), _) -> Zero()
eval: Add(Zero(), n) -> n
eval: Mul(Succ(m), n) -> Add(Mul(m, n), n)
eval: Add(Succ(m), n) -> Succ(Add(m, n))
```

While the concrete interpreter produces a single output depending on the input term, our abstract interpreter collects all alternative results systematically:

Concrete Stratego	Abstract Stratego

`Fail, Zero(), Succ(Zero())`
`Mul(Add(...), Zero()),`
`Add(Mul(Zero(), Zero()),..),`
`...`
	`Fail, Zero(), $Exp,`
	`Add(Mul($Exp,$Exp),$Exp),`
	`Succ(Add($Exp, $Exp))`

We still obtain a possible `Fail` result because the pattern match of `eval` is not exhaustive. For the other cases, the abstract interpreter yields one alternative result for each rewrite rule. When rewrite rules produce similar output terms like `Zero()` in our example, we share part of the corresponding analysis results.

So far, our evaluators only rewrite the input term once but do not normalize expressions to values. To do that, we have to add recursive calls of `eval` to our transformation, using the Stratego syntax `<f> t` for the application of transformation `f` to term `t`:

```
eval: Zero() -> Zero()
eval: Succ(n) -> Succ(<eval> n)
eval: Mul(Zero(), _) -> Zero()
eval: Mul(Succ(m),n) -> <eval> Add(Mul(m, n), n)
eval: Mul(e1,e2) -> <eval> Mul(<eval>e1, <eval>e2)
eval: Add(Zero(), n) -> <eval> n
eval: Add(Succ(m),n) -> Succ(<eval> Add(m, n))
eval: Add(e1,e2) -> <eval> Add(<eval>e1, <eval>e2)
```

This transformation recursively interprets an arithmetic expression and produces a Peano number (if the input expression is well-formed). In particular, we claim the transformation cannot produce intermediate terms containing the `Add` or `Mul` constructors. We would like our static analysis to confirm this hypothesis. Unfortunately, recursion also makes static analysis notoriously difficult. The problem is that our abstract interpreter must predict output terms for arbitrary input terms, on which it cannot recurse finitely. On the other hand, we want our abstract interpreter to yield a result in finite time. Hence, while our abstract interpreter starts recursively analyzing the program transformation, it has to abort recursion soon and overapproximate the result:

Concrete Stratego	Abstract Stratego

`Fail, Zero(), Succ(Zero())`
`Succ(Succ(Zero())), ...`
	`Fail, Zero(), Succ(Zero())`
	`Succ($Exp),Succ(Succ($Exp))`
	`$Exp`

This result is sound since nonterminal `Exp` subsumes all Peano numbers produced by the concrete interpreter. But this result is not very precise, because nonterminal `Exp` describes many terms that the transformation does not actually produce. Instead of `Exp`, we would much rather learn from the

analysis that the transformation is guaranteed to produce Peano numbers. Indeed, our analysis produced the following witnesses (all subsumed by `$Exp`):

```
Zero(), Succ(Zero()), Succ($Exp), Succ(Succ($Exp))
```

The witnesses indicate that the results appear to be Peano numbers, but there is no guarantee for that.

In the following we discuss the three main challenges for abstract interpretation of program transformations in some more detail: termination, soundness, and precision.

3 Challenge 1: Termination

An abstract interpreter must terminate in finitely many steps, no matter what. That is, an abstract interpreter must be a computable function over the transformation code, even though the transformation code may diverge. Abstract interpreters tend to analyze recursive programs through a fixpoint computation that converges after finitely many steps [Nielson et al. 1999]. This has triggered a lot of research about how to ensure a fixpoint is reached and how to speed up convergence.

To control termination of our abstract interpreter, we model its call stack explicitly: Every recursive call in the interpreter pushes a stack frame to the stack and every return pops the corresponding stack frame again. In order to ensure that the abstract interpreter indeed terminates, we adopt a technique developed by Van Horn and Might for abstract machines [Horn and Might 2010]. Where they allocated states of the abstract machine on a heap with finite address space, we allocate stack frames on such heap. Since the stack frames are persisted throughout the interpretation and the address space is finite, the interpreter will exhaust the address space eventually. While the interpreter can reuse previously used addresses, it must join the previous with the new results. This way, the interpreter can trade in precision for faster termination.

4 Challenge 2: Soundness

An abstract interpreter is sound if it predicts all possible results produced by the concrete interpreter. But is soundness an important property when analyzing program transformations? We argue that soundness is very important whenever analysis results are acted upon. The reason is that a sound analysis is reliable: It cannot "forget" any actual results and it cannot ignore corner cases. Thus, decisions formed on the basis of a sound abstract interpreter are actually meaningful.

The approach of abstract interpretation was designed by Cousot and Cousot to support the systematic construction of an abstract interpreter based on a concrete interpreter [Cousot and Cousot 1979]. However, proving soundness of an abstract interpreter is difficult and requires deep knowledge about order theory and Galois connections as well as reasoning about two interpreters in parallel (the concrete and the abstract one). For program transformations, proving soundness is particularly difficult because we have to reason about program properties rather than value properties — we have to reason about metaprograms rather than programs. For example, instead of reasoning about whether a variable holds a positive number after an assignment, we would have to reason about whether a transformation rule produces assignment code with that property. This is why our analysis targets a simpler property of program transformations, namely the syntactic shape of generated code. But even for this simpler property proving soundness is a serious endeavour: We failed to prove our abstract Stratego interpreter sound at first.

We are currently exploring a new style for defining abstract interpreters such that soundness proofs become easier. The key idea is to let the concrete and abstract interpreters share a parameterized implementation, such that a sound instantiation of the parameters yields a sound abstract interpreter for the concrete interpreter. To achieve the necessary parameterization, we are exploring what we call *arrow-based abstract interpreters*. Arrow-based abstract interpreters use *arrows* [Hughes 2000], a generalization of monads, to abstract from differences between the concrete and the abstract domain. We are currently using this approach to develop concrete and abstract interpreters for Stratego and prove them sound.

5 Challenge 3: Precision

The result of an abstract interpreter is only useful if it is sufficiently precise. For example, an abstract interpreter for Stratego that concludes a transformation produces "some term" would probably not be very useful. However, precision is necessarily limited given our previous requirements: termination and soundness. Therefore, the challenge is to find representations of program properties that are sufficiently precise yet soundly computable.

In Section 2, our abstract interpreter tried to predict the shape of terms produced by a transformation. We represented a transformation's possible output terms as a set of alternative terms with nonterminal symbols at places where the result was imprecise, as in `Succ($Exp)`. The termination requirement ensures that the set of alternatives always remains finite, since only finitely many alternatives can be explored before the analysis must terminate and conclude "some term".

We can change the precision of our abstract interpreter by changing the representation of program properties. In fact, the representation for the shapes of terms shown in Section 2 is already an improvement over the representation we used originally. Our original representation for shapes of terms did not distinguish between different nonterminals and used a single placeholder `*` for unknown terms, as in `Succ(*)`. However, we found that when analyzing target

languages with richer syntax based on many different nonterminals, a single placeholder loses too much information. For example, `Succ(*)` includes `Succ(Assign(...))` as well ass `Succ(MethodDec(...))`. Such imprecision causes ripple effects whenever generated code is also input to a transformation, that is, whenever a transformation produces intermediate terms. The problem with imprecise intermediate terms is that the imprecision in the input causes further imprecision in the output.

To improve the precision without losing soundness, we also had to change the concrete interpreter of Stratego. Specifically, Stratego is an untyped language that allows the construction of ill-formed terms such as `Succ(MethodDec(...))`. Because such terms witness bugs in many cases, in previous work [Erdweg et al. 2014] we proposed a dynamic type system for Stratego that prevents the construction of ill-formed terms altogether. Our representation of terms with nonterminals is only sound in this variant of Stratego.

In the future work, we want make our analysis significantly more precise. In particular, we would like the analysis to confirm that the evaluator for arithmetic expressions from Section 2 indeed only produces Peano numbers as output terms. To achieve this, we need a more precise representation of term shapes. Our plan is to use (regular) tree grammars [Aiken and Murphy 1991] for that purpose. Tree grammars describe sets of trees through recursive productions. For example, we can describe the Peano numbers as follows:

```
PN ::= Zero() | Succ(PN)
```

The main advantage compared to the representation of terms with nonterminals (`Succ($Exp)`) is that tree grammars are not required to reuse nonterminals from the target language. Instead, tree grammars can define new nonterminals for a subset of terms described by an existing nonterminal. For example, nonterminal `PN` describes a subset of the terms described by the existing nonterminal `Exp`. This can make our abstract interpreter significantly more precise. While it seems clear that tree grammars can still soundly approximate the actual output terms, termination is less certain. We will explore this in future work.

6 Evaluation

Accompanying this proposal, we created a prototype for an abstract interpreter of Stratego. The source code of the interpreter can be found online.[1] We evaluated the analysis results of our prototype for 4 program transformations: a PCF type checker and interpreter, a desugaring of the pretty notation for Haskell arrows [Hughes 2000], a normalization procedure for arrows [Liu et al. 2009], and a Go to JavaScript compiler. To evaluate the adequacy of the analysis results, we inspected terms in the result set of the analysis and classified them as true or false positives, depending on if there exists a concrete term that is transformed by the concrete Stratego interpreter to a term that fits the pattern of the abstract term. The analysis produced good results except for the *PCF* and *Go to JavaScript* case study, caused by imprecise tracking of aliases and by missing type information of terms. We intend to address these issues with extensions of the abstract interpreter in future work.

[1] https://github.com/svenkeidel/stratego-ai.git

7 Related Work

Abstract interpretation has been applied to program transformations in the past. Cousot and Cousot [2002] use abstract interpretation to assert that a transformation preserves the semantics of programs by showing that the abstract semantics is the same. Takai [2004] uses abstract interpretation to verify undecidable safety properties of term rewriting systems. And Mycroft [1982] uses abstract interpretation to justify program transformations that improve efficiency.

In this work, we use abstract interpretation to help developers reason about program transformations. In contrast to the mentioned work, we propose an abstract interpreter for program transformations that is independent of the semantics of the object language and computes syntactic properties of the meta language rather than semantic properties of the object language. This has the benefit that the same abstract interpreter can be used to analyze program transformations for all object languages and developers do not need to specify the semantics of their object language, which can be a huge effort. Furthermore, developers can deduce semantic properties of the object language from syntactic properties of the meta language by encoding an interpreter of the object language as a meta program.

8 Conclusion

We have proposed the abstract interpretation of program transformations in order to assist developers in reasoning about program transformations and the code they generate. Based on our experience on developing an abstract interpreter for the Stratego transformation language, we have presented and discussed three challenges we encountered: termination, soundness, and precision. While these challenges are common for any static analysis tool, they are specifically difficult to solve for static analyses of metapgrograms. We discussed how these challenges affected the development our abstract Stratego interpreter.

References

Alexander Aiken and Brian R. Murphy. 1991. Implementing regular tree expressions. In *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*. Springer-Verlag New York, Inc., New York, NY, USA, 427–447. http://dl.acm.org/citation.cfm?id=127960.128053

Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 238–252.

Patrick Cousot and Radhia Cousot. 1979. Systematic design of program analysis frameworks. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*. ACM, 269–282.

Patrick Cousot and Radhia Cousot. 2002. Systematic design of program transformation frameworks by abstract interpretation. In *ACM SIGPLAN Notices*, Vol. 37. ACM, 178–190.

Sebastian Erdweg, Vlad Vergu, Mira Mezini, and Eelco Visser. 2014. Modular Specification and Dynamic Enforcement of Syntactic Language Constraints. In *Proceedings of International Conference on Modularity (AOSD)*. ACM, 241–252.

David Van Horn and Matthew Might. 2010. Abstracting abstract machines. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*. 51–62.

John Hughes. 2000. Generalising monads to arrows. *Sci. Comput. Program.* 37, 1-3 (2000), 67–111.

Hai Liu, Eric Cheng, and Paul Hudak. 2009. Causal commutative arrows and their optimization. In *Proceedings of International Conference on Functional Programming (ICFP)*, Vol. 44. ACM, 35–46.

Alan Mycroft. 1982. Abstract interpretation and optimising transformations for applicative programs. (1982).

Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 1999. *Principles of program analysis*. Springer.

Toshinori Takai. 2004. A verification technique using term rewriting systems and abstract interpretation. In *RTA*, Vol. 3091. Springer, 119–133.

Eelco Visser, Zine-El-Abidine Benaissa, and Andrew P. Tolmach. 1998. Building Program Optimizers with Rewriting Strategies. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998*. 13–26.