Finding Bugs in Program Generators by Dynamic Analysis of Syntactic Language Constraints[1]

Sebastian Erdweg
TU Darmstadt, Germany

Vlad Vergu
TU Delft, Netherlands

Mira Mezini
TU Darmstadt, Germany

Eelco Visser
TU Delft, Netherlands

Abstract

Program generators and transformations are hard to implement correctly, because the implementation needs to *generically* describe how to construct programs, for example, using templates or rewrite rules. We apply *dynamic analysis* to program generators in order to support developers in finding bugs and identifying the source of the bug. Our analysis focuses on syntactic language constraints and checks that generated programs are syntactically well-formed. To retain a language's grammar as the unique specification of the language's syntax, we devised mechanisms to derive the analysis from the grammar. Moreover, we designed a run-time system to support the modular activation/deactivation of the analysis, so that generators do not require adaption. We have implemented the analysis for the Stratego term-rewriting language and applied it in case studies based on Spoofax and SugarJ.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Programming by contract; I.2.2 [*Automatic Programming*]: Program transformation; D.3.4 [*Processors*]: Runtime environments

General Terms Languages, Design

Keywords typesmart constructors; dynamic analysis; program transformation; generative programming; well-formedness checks; abstract syntax tree; Spoofax; SugarJ

1. Motivation

The application areas of program generators and program transformations are versatile and range from the implementation of program optimizations to the compilation from one language to another language, the injection of monitoring instructions, and the weaving of aspects. In our own work, we mostly use program transformations for the implementation of compilers for domain-specific languages in the context of language workbenches [5].

Unfortunately, it is difficult to implement program transformations correctly, because the implementation needs to *generically* describe how to construct programs, typically using templates or rewrite rules. Therefore, a transformation produces similar programs for related inputs that get answered by the same template or rewrite rule. Implementing transformations is difficult because each template or rewrite rule is responsible for a class of inputs and must translate each of the inputs to a well-behaved program.

Moreover, if a program transformation produces an incorrect result for some input, it is difficult to discover the origin of the mistake. Generally, there are two possibilities: Either the input was incorrect and should not have reached the transformation, or the transformation is incorrect and produced invalid code. But even if we know the input is correct and the transformation is to blame, it is still difficult to find out which part of the transformation exactly went wrong. To answer this question, we would have to identify which part of the result is incorrect, which part of the transformation constructed this part of the result, and what led the transformation to construct it incorrectly.

For example, consider the program transformations `compile1` and `compile2` displayed in Figure 1. Both transformations are implemented in the strategic term-rewriting language Stratego [10] and compile a lambda expression to one and the same anonymous Java class. The lambda expression binds variable `x` of type `atype` and has body `body` with result type `rtype`. From such a lambda expression, each transformation generates an anonymous class instance of interface `lambda.Function` and defines a public method `apply` that takes a parameter corresponding to the lambda-bound variable. The body of the generated method is defined by a recursive call of the transformation on the body of the lambda expression.

The second transformation `compile2` uses untyped abstract syntax (similar to s-expressions) to describe the generated code. As the abstract syntax of the target language Java is rather complicated, it is very easy to accidentally generate ill-formed code in `compile2`. For example, a missing `Id` tag around a name such as `"lambda"` or a forgotten `None` or `Some`, which are used to represent optional nodes. Such little mistakes are hard to trace and can entail severe problems that may break the rest of the processing pipeline, such as a static analysis or a pretty printer that expect syntactically well-formed code as input. The first transformation `compile1` avoids some of these mistakes by using concrete Java syntax in the generation template, which is parsed with an enriched Java grammar before the transformation is applied [9]. For example, the parser will automatically produce a well-formed abstract syntax tree for the qualified name `lambda.Function` that contains all necessary `Id` tags.

However, despite using concrete syntax and a parser, even `compile1` is not safe at all and can generate ill-formed code: When splicing external data into a generation template (designated by ~ in the template of `compile1`), the injected data must match the expected syntactic form. For example, both transformations assume that the types of lambda expressions have a Java encoding, because

[1] This is a demonstration paper accompanying the full article that has been published at Modularity'14 [6]

```
compile1 :
  Lambda(x, atype, rtype, body)
  ->
  |[ new lambda.Function<~atype, ~rtype>{
       public ~rtype apply(~atype ~x) { ~cbody; }
     }
  ]|
  where cbody := <compile> body
```

```
compile2 :
  Lambda(x, atype, rtype, body)
  ->
  NewInstance(
    None(),
    ClassOrInterfaceType(
      TypeName(
        PackageOrTypeName(Id("lambda")),
        Id("Function")),
      Some(TypeArgs([atype, rtype]))),
    [],
    Some(ClassBody(
      [MethodDec(
        MethodDecHead(
          [Public()], None(), rtype, Id("apply"),
          [Param([], atype, Id(x))], None()),
        cbody)])))
  where cbody := <compile> body
```

Figure 1. Compilation of a lambda expression to Java by transformation of the syntax tree, with and without using concrete syntax.

atype and rtype are injected unchanged into the generated Java code. Whether this is true or not cannot be answered by looking at Figure 1 alone. Instead, a (potentially global) data-flow analysis is necessary to statically determine the type encoding of the lambda expressions that are passed to compile as input. Similarly, compile assumes that the recursive compile call on the lambda-expression body results in a valid Java method body. Again, a data-flow analysis is required to ensure this statically. These examples only consider the syntactic structure of generated code; guaranteeing that generated code is well-typed would be even harder. In particular, due to Stratego's sophisticated language features (for example, rule overloading, generic traversals, or dynamically scoped rewrite rules), an efficient static analysis would be hard to design and most likely very specific to the Stratego language and not reusable for other metaprogramming systems. Stratego is not the only metaprogramming system that fails to guarantee the well-formedness of generated code. In particular, metaprograms written in similarly flexible programming languages such as Python, Ruby, or JavaScript exhibit the same problem.

2. Dynamically analyzing program generators

We propose to apply dynamic analysis to program generators in order to identify incorrect transformation results and the origin of the mistake. Specifically, a dynamic analysis monitors the execution of a program transformation and interrupts the execution as soon as it identifies an error. The analysis can then report details about the origin of the error based on the monitored execution state and the involved input data.

For this work, we designed and implemented a dynamic analysis that checks if a transformation produces well-formed syntax trees. That is, whenever the a transformation produces a (fragment of a) syntax tree, our analysis checks whether the tree adheres to the context-free syntax of the target language. When the analysis recognizes an ill-formed tree, it can point out the transformation that produced the tree.

For example, the analysis checks that the transformation shown in Figure 1 calls the constructor Param in the generated method header with a valid type and identifier. The analysis triggers as soon as the an ill-formed Param node is constructed and, thus, rejects the construction if the parameter name is not wrapped in an Id syntax-tree node or the parameter type atype is not a well-formed syntax tree representing a Java type.

In fact, the analysis monitors the construction of every syntax-tree node and checks whether the arguments are valid for the given node. To do so, it computes the syntactic sort of the arguments and checks whether they conform to the constructor. To reduce performance overhead, the analysis caches the syntactic sort of a tree on construction of that tree, so that the sort of arguments is always directly available. For example, when checking the Param construction, the analysis does not have to recurse on the arguments to retrieve their syntactic sort, but it can retrieve the sorts from the cache. This way, we were able to limit the analysis to a constant run-time overhead per construction of a syntax-tree node.

Typesmart constructors. We also propose an implementation strategy for realizing dynamic analyses of program generators, called typesmart constructors. A typesmart constructor is a conventional function that acts like a regular constructor and creates tree nodes. However, in contrast to a regular constructor, a typesmart constructor may reject the creation of a node if this would violate a language-specific invariant. For example, a typesmart version of the constructor Param from the example above would reject the construction if the parameter name is not wrapped in an Id syntax-tree node or the parameter type atype is not a well-formed syntax tree representing a Java type.

Consider a constructor C with signature

```
C :: A -> B.
```

A typesmart constructor for C is any function f with signature

```
f :: A -> (fail or B*)
```

that satisfies

```
f(a) = fail  or  f(a) = C(a).
```

Here, B* denotes the type B augmented with annotations of auxiliary data such as the syntactic sort or the type of a term. We assume term equality (=) ignores annotations. Accordingly, a typesmart constructor for C behaves exactly like C except that it may fail or annotate auxiliary data to the constructed value.

Typesmart constructors can be used to enforce invariants about constructed data. For example, here are two typesmart list constructors that enforce that all list elements are even integers:

```
nil() = Nil()
cons(x, xs) =
  if x % 2 == 0
   then Cons(x, xs)
   else fail
```

By calling typesmart constructors in place of regular constructors, the developer of a program transformation can activate the corresponding dynamic analysis. However, we also devised support for modularly and transparently activating typesmart constructors, as we discuss in the subsequent section.

3. Modular specification and modular enforcement of well-formedness

In this work, we particularly focused on how to modularly specify and modularly enforce syntactic well-formedness with typesmart constructors. Typically, a language's syntax is specified centrally by a grammar. It is bad practice to duplicate such specification because this impedes consistency and maintainability. Instead, we want to retain the grammar as a modular specification of a language's syntax. Conversely, manually applying typesmart constructors is cross-cutting the whole transformation: Every occurrence of a regular constructor must be replaced by a typesmart constructor. Thus, the well-formedness of generated code relies on the users' discipline to actually call typesmart constructors in place of regular constructors. Especially, when using third-party libraries, such discipline cannot be expected.

To avoid replicating information of the grammar in the definition of typesmart constructors, we devised a transformation that *derives* typesmart constructors from the grammar automatically. This way, if the grammar changes, we can derive updated typesmart constructors easily. Our transformation expects the grammar to be expressed in SDF2 [8], which uses scannerless generalized LR parsing. Since SDF2 does neither require constructor names to be unique nor that every production has a constructor, the transformation has to deal with ambiguous constructor usage. To this end, the derived typesmart constructors use a subsort relation and allow a tree to have multiple sorts. Details can be found in the full paper [6].

To support modular activation and deactivation of typesmart constructors, we integrated support for typesmart constructors into the run-time system of the transformation language Stratego [10]. Specifically, we modified the way Stratego terms are constructed such that a call to a regular constructor is *always* redirected to the corresponding typesmart constructor. Since this redirection is modularly defined, automatic, and transparent to users, we obtain the following advantages: (i) transformations do not have to be changed in any way, (ii) transformations can rely on the global guarantee that all abstract syntax trees represent syntactically well-formed programs during the whole execution, and (iii) dynamic checks can be modularly activated and deactivated. Again, further details can be found in the full paper [6].

4. Application

We applied our dynamic analysis for syntactic well-formedness within Spoofax [7], SugarJ [1, 2], and SugarHaskell [3, 4]. Spoofax is a language workbench for agile development of external textual languages with IDE support. SugarJ is an extensible language that encapsulates language extensions as regular base-language modules that can be activated via import statements. Both Spoofax and SugarJ use Stratego as underlying term transformation language with which a language developer/extender can define static analyses, program semantics, and semantic editor services.

Spoofax. In Spoofax, we applied typesmart constructors in a project that implements a compiler for a subset of Java called MiniJava. The MiniJava compiler is a Stratego program that transforms MiniJava trees into their equivalent counterpart written in the Jasmin assembler language [1] for the Java Virtual Machine. We tested the MiniJava compiler by applying it to 233 programs written in MiniJava. To our surprise, this gradually uncovered more than 20 bugs in the Jasmin generator, which we repaired. The uncovered defects caused syntactically incorrect Jasmin ASTs to be generated by the compiler. The majority of violations involved missing constructors that wrap references to classes, fields, and labels. For example, we changed the compiler as follows:

```
Reference("java/io/PrintStream")
  ⇝ Reference(CRef("java/io/PrintStream"))

GOTO(end)
  ⇝ GOTO(LabelRef(end))
```

When working with abstract syntax trees, this is a typical problem: The abstract syntax requires more intermediate nodes than seems necessary for a programmer. Therefore, it is easy to forget some of these nodes, such as LabelRef. Note that using concrete syntax in the generation template [9] would only have resolved the former violation but not the latter violation, because the sort of end is unknown at compile time.

Another significant part of the morphological errors were caused by mismatching types, such as integers used instead of strings, and ill-placed or missing constructors:

```
ALOAD(n)
  ⇝ ALOAD(VarNum(<int-to-string> n))

JBCVarDecl(
  VarNum(n), x, <to-jbc> t,
  LabelRef(START()), LabelRef(END()))
⇝
JBCVarDecl(
  <int-to-string> n, x, JBCFieldDesc(<to-jbc> t),
  LabelRef(START()), LabelRef(END()))
```

In the MiniJava compiler, the generated Jasmin code is forwarded to a rather permissive pretty-printer that only locally applies formatting rules that do not capture nor rely on the hierarchical structure of the tree. We suspect that the errors we found remained hidden until now because the pretty-printer accepts these ill-formed syntax trees and emits syntactically correct concrete Jasmin syntax. In different application scenarios, the bugs we found could have been severe. Especially, forwarding ill-formed code to another program transformation, for example a byte-code verifier or optimizer, may lead these tools to fail. Such bugs are very hard to track down, because a developer needs to manually retrace the data flow of the generated program to discover where the ill-formed term was originally constructed. Typesmart constructors reject ill-formed programs right away when they are constructed. This provides precise and early feedback to developers.

SugarJ and SugarHaskell. In SugarJ and SugarHaskell, we applied typesmart constructors for validating language extensions of Java and Haskell (transformations are implemented in Stratego). This led to the discovery of a number of bugs in previously developed and tested language extensions. Similar to the MiniJava compiler, we found a few instances of missing constructor applications that were supposed to wrap expression literals, variable symbols in expressions, etc. But we also found more complicated defects.

For example, in a SugarHaskell language extension that introduces special syntax for "idiomatic brackets", we found a bug related to constructor and variable symbols in Haskell. The Haskell grammar distinguishes constructor symbols (starting with an upper-case character) from variable symbols (starting with a lower-case character). One transformation failed to retain this distinction. We had to rewrite the production of the language extension and the desugaring to fix the issue:

```
"(|" Exp Qop Exp "|)" -> Exp {"IdiomBrack"}
  ⇝ "(|" Exp Qvarsym Exp "|)" -> Exp {"VarIdiomBrack"}

<apply-effect> (BinCon(op), [e1, e2])
  ⇝ <apply-effect> (BinOp(op), [e1, e2])
```

The first change restricts the production for idiomatic brackets to only permit variable symbols and to prohibit constructor symbols. The second change fixes the generation template, which was declaring that the parsed symbol is a constructor whereas it in fact is a variable. Would we break this distinction between constructors and variables (as the original code did), this may have far-reaching

[1] http://jasmin.sourceforge.net

consequences. For example, subsequent optimizations may assume that constructor calls can be executed cheaply, whereas regular function calls should be inlined or are subject to further optimization if they are recursive. An optimization that transforms the program accordingly would have failed. Typesmart constructors notified us of the error immediately when the illegal program fragment was generated, instead of silently failing.

We also applied typesmart constructors to three Java extensions implemented with SugarJ: tuple notation, lambda expressions, and literal XML. We did not discover any additional syntactic errors in code generated from these extensions for our test programs. Since we only used a handful of test inputs for our language extensions, it might well be that the test coverage was too low. Alternatively, the transformations indeed are safe and produce well-formed programs.

5. Conclusion

To support developers of program transformations, we propose to apply dynamic analyses that check the well-formedness of generated code. So far we focused on syntactic well-formedness and implemented a dynamic analysis that guarantees that generated code adheres to the grammar of the target language. We propose typesmart constructors as an implementation technique for dynamic analyses of program generators. Application of typesmart constructors in existing transformations indicates their effectiveness to reveal bugs, even within well-tested code.

To enable the modular specification and enforcement of well-formedness checks, we derive typesmart constructors from a language's grammar and provide support for transparent activation and deactivation of typesmart constructors in the run-time system of Stratego. In future work, we plan to explore the application of typesmart constructors to check for semantic properties such as well-typedness.

References

[1] S. Erdweg. *Extensible Languages for Flexible and Principled Domain Abstraction*. PhD thesis, Philipps-Universiät Marburg, 2013.

[2] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. SugarJ: Library-based syntactic language extensibility. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 391–406. ACM, 2011.

[3] S. Erdweg and F. Rieger. A framework for extensible languages. In *Proceedings of Conference on Generative Programming and Component Engineering (GPCE)*, pages 3–12. ACM, 2013.

[4] S. Erdweg, F. Rieger, T. Rendel, and K. Ostermann. Layout-sensitive language extensibility with SugarHaskell. In *Proceedings of Haskell Symposium*, pages 149–160. ACM, 2012.

[5] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. Vergu, E. Visser, K. van der Vlist, G. Wachsmuth, and J. van der Woning. The state of the art in language workbenches. In *Proceedings of Conference on Software Language Engineering (SLE)*, volume 8225 of *LNCS*, pages 197–217. Springer, 2013.

[6] S. Erdweg, V. Vergu, M. Mezini, and E. Visser. Modular specification and dynamic enforcement of syntactic language constraints. In *Proceedings of International Conference on Modularity (AOSD)*, 2014. to appear.

[7] L. C. L. Kats and E. Visser. The Spoofax language workbench: Rules for declarative specification of languages and IDEs. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 444–463. ACM, 2010.

[8] E. Visser. A family of syntax definition formalisms. Technical Report P9706, Programming Research Group, University of Amsterdam, August 1997.

[9] E. Visser. Meta-programming with concrete object syntax. In *Proceedings of Conference on Generative Programming and Component Engineering (GPCE)*, volume 2487 of *LNCS*, pages 299–315. Springer, 2002.

[10] E. Visser, Z.-E.-A. Benaissa, and A. P. Tolmach. Building program optimizers with rewriting strategies. In *Proceedings of International Conference on Functional Programming (ICFP)*, pages 13–26. ACM, 1998.