Modular Specification and Dynamic Enforcement of Syntactic Language Constraints when Generating Code

Sebastian Erdweg
TU Darmstadt, Germany

Vlad Vergu
TU Delft, Netherlands

Mira Mezini
TU Darmstadt, Germany

Eelco Visser
TU Delft, Netherlands

Abstract

A key problem in metaprogramming and specifically in generative programming is to guarantee that generated code is well-formed with respect to the context-free and context-sensitive constraints of the target language. We propose *typesmart constructors* as a dynamic approach to enforcing the well-formedness of generated code. A typesmart constructor is a function that is used in place of a regular constructor to create values, but it may reject the creation of values if the given data violates some language-specific constraint. While typesmart constructors can be implemented individually, we demonstrate how to derive them automatically from a grammar, so that the grammar remains the sole specification of a language's syntax and is not duplicated. We have integrated support for typesmart constructors into the run-time system of Stratego to enforce usage of typesmart constructors implicitly whenever a regular constructor is called. We evaluate the applicability, performance, and usefulness of typesmart constructors for syntactic constraints in a compiler for MiniJava developed with Spoofax and in various language extensions of Java and Haskell implemented with SugarJ and SugarHaskell.

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. Introduction and motivating example

Metaprograms process other programs as data, for example, to realize program optimizations, to compile a program to another language, or to inject monitoring code. A traditional problem of metaprogramming is to guarantee that generated code is well-formed according to the syntax and type system of the target language. Such a guarantee is important because it provides valuable feedback to developers of metaprograms and rules out a whole

MODULARITY '14, April 22–26, 2014, Lugano, Switzerland.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2772-5/14/04...$15.00.
http://dx.doi.org/10.1145/2577080.2577089

```
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.

class of errors that would lead to subsequent metaprogramming tools to fail or even to unsound behavior of the generated program at its run time.

For example, consider the program transformations compile1 and compile2 displayed in Figure 1. Both transformations are implemented in the strategic term-rewriting language Stratego [27] 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 [26]. 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 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.

We propose *dynamic checking* of language-specific invariants on generated code at construction time using *typesmart constructors*. A typesmart constructor is a conventional function that acts like a regular constructor and creates data values. However, in contrast to a regular constructor, a typesmart constructor may reject the creation of a value if this would violate a language-specific invariant. For example, a typesmart version of the constructor Param used in the generated method header in Figure 1 would reject the construction of parameters where the parameter name is not wrapped in an Id syntax-tree node or where the parameter type atype is not a well-formed syntax tree representing a Java type. To communicate metadata about a program (such as a syntactic sort or type), typesmart constructors read and write annotations on the abstract syntax tree. For example, the typesmart version of constructor Param would query the annotation of atype to identify its syntactic sort.

In this paper, we particularly focus on the syntactic well-formedness of generated programs and how to *modularly specify and 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: Typesmart constructors should neither duplicate information of the grammar, nor should the grammar be coupled to typesmart constructors. To this end, we devised a transformation that extracts the conditions for syntactic well-formedness from a grammar and generates corresponding typesmart constructors automatically. For enforcing the generation syntactically well-formed programs , the application of typesmart constructors in place of regular constructors should be transparent to developers of program transformations and should not require any change to existing transformations. To achieve this, we designed runtime-system support for typesmart constructors that does not rely on the user's discipline and that can be activated and deactivated modularly. In summary, we make the following contributions:

- We propose typesmart constructors for dynamically checking language-specific well-formedness criteria on generated code. Typesmart constructors are applicable in any metaprogramming system
- We designed and implemented a transformation that automatically derives typesmart constructors from a language's syntax definition to dynamically check the syntactic structure of generated code.
- We incorporated support for typesmart constructors in the runtime system of Stratego to transparently enforce the usage of typesmart constructors in place of regular constructors. This establishes the global invariant that all generated code is well-formed at all times, and transformations do not have to be adapted.
- We evaluate the applicability, performance, and usefulness of typesmart constructors for syntactic constraints by investigating their application in a compiler for MiniJava written in Stratego and in various language extensions of Java and Haskell implemented in SugarJ and SugarHaskell. Using typesmart constructors, we found 27 bugs in existing, tested program transformations.

In this paper, we explore the standard trade-off between static and dynamic analyses: On the one hand, dynamic analyses are easier to define and understand than static data-flow analyses (in particular, if sophisticated metalanguage features should be supported). On the other hand, static analyses deliver feedback at an earlier stage and do not entail any runtime overhead. We see this work as an initial step to support sophisticated language-specific invariants in metaprogramming systems with sophisticated metalanguage features. We envision future work on hybrid analyses that reduce the runtime overhead and deliver feedback earlier when possible, and on dynamic analyses that enforce semantic properties of a language at program-generation time.

2. Typesmart constructors

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
```

Similar functionality is supported by contracts on structures in Racket [21]. Indeed, typesmart constructors can be seen as a form of flat contracts [11] for constructors. However, in contrast to contracts, typesmart constructors are allowed to modify the resulting value by adding annotations. For example, we can use annotations to efficiently ensure that the tail of the constructed list consists of even integers as well:

```
nil() = Nil()
cons(x, xs) =
  if x % 2 == 0 && get-anno(xs, "list-of-even") == True
   then put-anno(Cons(x, xs), "list-of-even", True)
   else fail
```

Functions get-anno and put-anno read and write a named annotation of a term, respectively. Without annotations, the typesmart constructor cons would have to recheck the tail of the list each time another element is added, which would change the asymptotic complexity of cons from constant to linear.

An important property of tree-like terms is that they are always built bottom-up. Accordingly, the arguments of a constructor have themselves been previously constructed by other constructors, which may have been typesmart or regular ones. We use annotations in typesmart constructors for three purposes:

- To ensure that the arguments of a typesmart constructor have themselves been constructed by a typesmart constructor.
- To ensure that all required invariants have been enforced on arguments of a typesmart constructor.
- And more generally, to provide a channel of communication from children to parents during the construction of terms.

We can leverage these properties of typesmart constructors to enforce invariants about terms. For the special and important case where we generate programs, we can use typesmart constructors to enforce language-specific constraints. In this paper, we focus on the enforcement of syntactic language constraints.

3. Typesmart constructors for syntactic language constraints

Let us consider a simple language with variables, expressions, and statements (names in curly braces denote the desired constructor names):

```
Var ::= String    {Var}
Exp ::= Var       {VarExp}
      | Integer   {Num}
      | Exp + Exp {Add}
Stm ::= Var = Exp {Assign}
      | Stm; Stm  {Seq}
```

We can define the typesmart constructors for this language as shown in Figure 2. Each constructor checks that its arguments have the appropriate sort. For example, the constructor for assignments requires that the first argument is a variable and the second argument is an expression. An assignment itself is of sort statement. Essentially, the typesmart constructors of Figure 2 ensure that all generated programs adhere to the above grammar.

In statically typed programming languages, the syntactic structure of our simple language can be easily encoded using, for example, algebraic data types. This would statically guarantee that all generated programs adhere to the grammar. We are interested in achieving similar guarantees in dynamically typed languages with sophisticated language features such as generic traversals. Enforcing syntactic constraints with typesmart constructors poses the following challenges:

```
var(v) =
  if is-string(v)
   then put-anno(Var(v), "sort", "Var")
   else fail
varexp(v) =
  if get-anno(v, "sort") == "Var"
   then put-anno(VarExp(v), "sort", "Exp")
   else fail
num(n) =
  if is-integer(n)
   then put-anno(Num(n), "sort", "Exp")
   else fail
add(e1, e2) =
  if get-anno(e1, "sort") == "Exp" &&
    get-anno(e2, "sort") == "Exp"
   then put-anno(Add(e1, e2), "sort", "Exp")
   else fail
assign(v, e) =
  if get-anno(v, "sort") == "Var" &&
    get-anno(e, "sort") == "Exp"
   then put-anno(Assign(v, e), "sort", "Stm")
   else fail
seq(s1, s2) =
  if get-anno(s1, "sort") == "Stm" &&
    get-anno(s2, "sort") == "Stm"
   then put-anno(Seq(s1, s2), "sort", "Stm")
   else fail
```

Figure 2. Typesmart syntax constructors for a simple language with variables, expressions, and statements.

- Not all terms correspond to user-defined sorts. For example, no user-defined sort exists for lists [a,b,c], tuples (a,b,c), or optional elements None() and Some(t). Moreover, the constructors for lists, tuples, and optional terms are inherently polymorphic. For example, for any nonterminal S, the constructor Some generates a term of sort S? given a term of sort S.
- Many languages allow the injection of terms without delimiting constructors. For example, it is often the case that any valid term of sort Variable can be used as a valid term of sort Expression. Typesmart constructors need to consider such injections when checking the sorts of the constructor arguments.
- If a constructor occurs multiple times in a grammar (for example, due to language composition), the constructor may be able to produce terms of alternative sorts for the same arguments. Technically, such a term would be valid with respect to all alternative sorts and no preference can be safely made at this point.

We resolve these issues by (i) built-in support for primitive polymorphic term constructors for lists, tuples, and optionals, (ii) explicit support for a subsort relation that corresponds to term injections, and (iii) alternative result sorts for constructor calls that satisfy multiple productions simultaneously (similar to union types). We provide these features as a library that implementors of typesmart constructors for syntactic constraints can use. In the following, we present the exact definition of this library.

3.1 A library for checking syntax sorts

We define an auxiliary library that provides a high-level function has-sort(t, s) -> Bool to check whether term t has sort s. Function has-sort supports primitive polymorphic terms, alternative sorts, and checks a term with respect to subsorting. To this end, has-sort is configurable with a user-defined subsort relation.

We define function has-sort formally via inference rules. Figure 3 shows the abstract representation of terms t and sorts s that we use in the definition of has-sort. A term is either constructed

$$\begin{array}{rll|rl}
c ::= & \mathit{string} & \text{constructors} & & \\
t ::= & c(t,\dots,t) & \text{constructed} & s ::= & \mathit{string} \\
\mid & \mathit{string} & \text{string literals} & \mid & \mathsf{String} \\
\mid & [t,\dots,t] & \text{lists} & \mid & \mathsf{List}(s) \\
\mid & (t,\dots,t) & \text{tuples} & \mid & \mathsf{Tuple}(s,\dots,s) \\
\mid & \mathsf{None} \mid \mathsf{Some}(t) & \text{optionals} & \mid & \mathsf{Option}(s) \\
 & & \text{alternatives} & \mid & \mathsf{Alt}(s,s)
\end{array}$$

Figure 3. Abstract syntax for terms t and sorts s.

$$\textsc{String}\ \frac{\mathit{is\text{-}string}(t)}{t : \mathsf{String}} \qquad \textsc{List}\ \frac{t_1 : s \quad \cdots \quad t_n : s}{[t_1,\dots,t_n] : \mathsf{List}(s)}$$

$$\textsc{Tuple}\ \frac{t_1 : s_1 \quad \cdots \quad t_n : s_n}{(t_1,\dots,t_n) : \mathsf{Tuple}(s_1,\dots,s_n)}$$

$$\textsc{Opt1}\ \frac{}{\mathsf{None} : \mathsf{Option}(s)} \qquad \textsc{Opt2}\ \frac{t : s}{\mathsf{Some}(t) : \mathsf{Option}(s)}$$

$$\textsc{Alt1}\ \frac{t : s_1}{t : \mathsf{Alt}(s_1, s_2)} \qquad \textsc{Alt2}\ \frac{t : s_2}{t : \mathsf{Alt}(s_1, s_2)}$$

$$\textsc{Anno}\ \frac{\mathit{get\text{-}anno}(\texttt{"sort"}, t) \equiv s}{t : s}$$

$$\textsc{Sub}\ \frac{t : s_1 \quad s_1 <: s_2}{t : s_2} \qquad \textsc{Trans}\ \frac{s_1 <: s_2 \quad s_2 <: s_3}{s_1 <: s_3}$$

Figure 4. Definition of function has-sort, written as $t : s$.

through a constructor application, or it is a list, a tuple, or an optional term None or Some(t). A sort is either a user-defined sort or represents a list, a tuple, an option, or an alternative of two other sorts. We define function has-sort in Figure 4, where we write a call has-sort(t, s) in relational style $t : s$.

The definition of has-sort is mostly straight-forward. A literal string has sort String, a list with elements of sort s has sort List(s), an Option sort describes terms None and Some, a tuple has sort Tuple with matching component sorts. For alternatives Alt(s_1, s_2), it suffices if the term has either sort s_1 or sort s_2. If a term already has a sort annotation (that is, it was constructed by a typesmart constructor), we compare the annotated sort to the required one s in rule ANNO. To support terms annotated with alternative sorts, we use a special equivalence relation $\equiv$ that has special treatment for alternative sorts:

$$\textsc{Eq1}\ \frac{s_1 \equiv s}{\mathsf{Alt}(s_1, s_2) \equiv s} \qquad \textsc{Eq2}\ \frac{s_2 \equiv s}{\mathsf{Alt}(s_1, s_2) \equiv s}$$

That is, if the constructed term adheres to multiple sorts, it suffices if one of them matches the required sort s. Otherwise, the equivalence relation checks for syntactic equality.

Finally, function has-sort as defined in Figure 4 employs a subsumption rule SUB: If a term t has sort s_1 and sort s_1 is a subsort of sort s_2, then t also has sort s_2. Here, the subsort relation corresponds to valid term injections. For example, if our language allows the occurrence of variables in expressions Exp ::= Var, then Var would be subsort of Exp (Var <: Exp). The subsort relation is transitive as declared by rule TRANS. Apart from that, the subsort relation is unspecified and can be configured according to the syntactic constraints of a language, as we show in the following example.

3.2 Example language and example term construction

Using the function has-sort and the subsort relation, we can implement typesmart constructors that comply with the challenges from the beginning of this section. Let us consider the following small example language that highlights issues we observed in grammars of real-world languages.

```
Var    ::= String                  {Var}
Exp    ::= Var
         | "[" Exp* "]"            {Exps}
Param  ::= String                  {Var}
Proc   ::= "proc" Param? "=" Exp   {Proc}
```

Programs of this language are described with primitive terms for lists (Exp*) and optionals (Param?), perform term injection of variables into expressions, and variables and parameters share the constructor name Var and can be used interchangeably. Using the library from the previous subsection, we can define typesmart constructors for this language as follows:

```
var(v) =
  if is-string(v)
    then put-anno(Var(v), "sort", Alt("Var","Param"))
    else fail
exps(xs) =
  if has-sort(xs, List("Exp"))
    then put-anno(Exps(xs), "sort", "Exp")
    else fail
proc(p, e) =
  if has-sort(p, Option("Param")) && has-sort(e, "Exp")
    then put-anno(Proc(p,e), "sort", "Proc")
    else fail
Var <: Exp = True
```

These typesmart constructors require all features our has-sort library provides: primitive terms, term injection, and alternative sorts. Let us illustrate these features by constructing the term:

```
Proc(Some(Var("x")), Exps([Var("y")]))
```

1. The term Var("x") gets assigned sort Alt("Var","Param") since "x" is a string.
2. The term Some(Var("x")) has sort Option(Alt("Var","Param")).
3. The term Var("y") gets assigned sort Alt("Var","Param") since "y" is a string.
4. The term [Var("y")] has sort List(Alt("Var","Param")).
5. The term Exps([Var("y")]) gets assigned sort "Exp" since the argument to Exps is a list of expressions as checked by has-sort(xs, List("Exp")): According to the definition of has-sort in Figure 4, this holds if all elements of xs have sort Exp. However, the only element of xs is Var("y"), which has a sort that does not match: Alt("Var","Param") $\not\equiv$ "Exp". Therefore, we cannot use rule ANNO just yet. Instead, we first have to apply rule SUB using Var <: Exp to get Var("y") : "Var", which we can discharge using ANNO and Alt("Var","Param") $\equiv$ "Var".
6. The full term Proc(Some(Var("x")), Exps([Var("y")])) has sort "Proc": We first check the optional parameter, which succeeds due to rules OPT2, ANNO, and Alt("Var","Param") $\equiv$ "Param". Then we check the body Exps(...), which succeeds immediately using ANNO since it has a sort annotation Exp as required.

From this example term construction, we observe that at the time we constructed the Var terms, we did not yet know whether we need the terms as variables or as parameters. Since the Var terms adhere to the structure of both sorts (the abstract syntax overlaps), we marked the terms with an alternative sort that allows their usage in either contexts. Indeed, this was required in our example, because Var("x") was used as a parameter whereas Var("y") was used as a variable. Furthermore, we observe that due to alternative sorts and subsorts, there is quite a number of different cases to consider when checking the sort of an argument. For example, for the body of a procedure, valid argument sorts are Exp, Var, Alt("Var","Param"), but not Param and not Proc. Our library function has-sort greatly reduces the effort of writing typesmart constructors by taking care of all alternatives for saturating a sort requirement.

Nevertheless, the implementation of manual typesmart constructors is tedious, does not reflect the enforced language constraints declaratively, and duplicates knowledge about the syntax of the target language. In the subsequent section, we show that the grammar of the target language in fact can serve as a declarative and modular specification for typesmart constructors that enforce the corresponding syntactic constraints.

4. Deriving typesmart syntax constructors

A grammar should be the ultimate reification of all syntactic constraints a language possesses. Typesmart constructors enforce the syntactic constraints of a language dynamically while programs of the language are generated. In the previous section, we required the manual implementation of typesmart constructors. This limited modularity of the language definition since knowledge about the syntactic constraints is duplicated in the grammar and typesmart constructors. Moreover, the procedural implementation of typesmart constructors is neither declarative nor simple (as we shall see below).

We have developed a transformation that automatically derives typesmart constructors from a language's grammar. Our transformation assumes a grammar defined with SDF [25]. From such a grammar, it generates typesmart constructors as follows.

Regular productions with constructor (`N ::= rhs {C}`). From the right-hand side rhs, we extract the expected argument sorts (as of Figure 3) of the constructor C. First, all lexical tokens are removed; they are not part of the abstract syntax tree that we are constructing. For the rest of rhs, a nonterminal name is its own sort, an optional expression e? has sort Option of the sort of e, a list e* has sort List of the sort of e, and so on.

We generate exactly one typesmart constructor for each constructor for a given arity (number of arguments sorts). The typesmart constructor uses function has-sort (Section 3.1) to test each actual argument for conformance to the expected argument sort. If multiple productions define the same constructor C with the same arity, the single typesmart constructor we generate will test the actual arguments against each list of expected argument sorts. If the actual arguments conform to no production, the typesmart constructor fails and rejects the building of an ill-formed term. If the actual arguments conform to exactly one production, the typesmart constructor returns a term of the result sort of this production. If the actual arguments conform to multiple productions, the typesmart constructor returns a term of an alternative sort $\mathsf{Alt}(s_1, s_2)$ with one alternative per successful production.

For example, for the productions

```
A ::= "(" A A ")" {C}
B ::= "{" B B "}" {C}
```

we generate a single typesmart constructor

```
c(x,y) =
  if has-sort(x, "A") && has-sort(y, "A")
   then if has-sort(x, "B") && has-sort(y, "B")
         then put-anno(C(x,y), Alt("A", "B"))
         else  put-anno(C(x,y), "A")
   else if has-sort(x, "B") && has-sort(y, "B")
         then put-anno(C(x,y),"B")
         else  fail
```

If the constructor arguments fit both productions, the resulting tree has an alternative sort. Otherwise, if the arguments only fit one production, the resulting tree has the sort this production. If the arguments match neither production, the typesmart constructor fails.

As the number of constructor arguments and the number of productions with equally-named constructors grows, the implementation of the corresponding typesmart constructor becomes tedious and error-prone. For example, the largest typesmart constructor generated for Java has 7 arguments and comprises more than 30 lines of code. Since our transformation generates these constructors from a grammar automatically, language developers do not have to bother with the implementation details of typesmart constructors.

Injection productions (`N ::= A`). We use injection productions to define the subsort relation, over which function has-sort is parameterized. For each injection production N ::= A, we add the fact (A <: N) to the definition of the subsort relation (<:). In addition, we augmented function has-sort to check for cyclic injections when using rule SUB. Otherwise, an infinite loop A <: B <: A <: ... would make our dynamic analysis loop.

Lexical productions (`N ::= [a-z]+`). SDF supports lexical productions, which do not yield nodes in the abstract syntax tree. Instead, a lexical production always yields a string literal representing the parsed fragment of the input string. To support lexical productions in typesmart constructors, we install the nonterminal of a lexical production as a subsort of the sort String (N <: String).

Renamed nonterminals. SDF supports the renaming of nonterminals when importing a module. This provides a means for avoiding name clashes between nonterminals defined in independent modules. For example, one module may define a production Exp ::= Exp "+" Exp {Plus}. Another module may want to use a renamed version JavaExp of nonterminal Exp like this: Body ::= JavaExp {Body}. However, this is problematic for the typesmart constructor of Body, which expects a single argument of the renamed sort JavaExp. However, the concrete argument will have sort Exp. Essentially, the original and the renamed nonterminal are equivalent: We can use either original or renamed sort where the other one is expected. We encode this equivalence using the subsort relation: For each renaming of N to N', we install N as a subsort for N' and vice versa (N <: N' and N' <: N). The check for cyclic injections applies here as well to prevent infinite looping.

Implementation. We implemented the derivation of typesmart constructors as a Stratego transformation that accepts an SDF grammar as input and outputs Stratego code that implements the generated typesmart constructors. For evaluation, we applied our transformation to a composed grammar consisting of productions for SDF and Stratego, which results in 5595 source lines of Stratego code implementing a total of 361 typesmart constructors and installing a total of 262 subsort relationships. Furthermore, we applied our transformation to a grammar for Java, which results in 3989 source lines of Stratego code implementing a total of 206 typesmart constructors and installing a total of 307 subsort relationships. Given that the grammar provides a full specification of the syntactic constraints of a language, all of this generated code provides useful but redundant information. We would not have wanted to implement typesmart constructors for these languages by hand, and our generator provides required tool support for automatically deriving typesmart constructors from a grammar. Especially, this permits the modular evolution of the grammar, since typesmart constructors can simply be regenerated.

5. Run-time support for typesmart syntax constructors in Stratego

The idea of typesmart constructors is applicable in any language that provides an explicit notion of construction. However, manually applying typesmart constructors is cross-cutting the whole transformation: Every occurrence of a regular constructor must be replaced by a typesmart constructor. Moreover, 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 remedy this situation, we integrated support for typesmart constructors into the runtime system of Stratego. 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. In this section, we describe the augmented architecture of the Stratego runtime system.

Stratego [27] is a declarative language for the transformation of syntax trees. Conceptually, the runtime system of Stratego can be separated into two components: (i) the abstract syntax trees, generally referred to as *terms*, and (ii) the rewriting rules that transform terms. Stratego terms are immutable data objects that may have attachments. Attachments are used to store metadata about a term as required in different execution contexts, such as parent attachments pointing to the parent of a node or origin information establishing a link back from the result of a transformation to its input.

The Stratego runtime system produces terms via *term factories*. A term factory provides methods for the construction of different term types (integer, string, list, tuple, constructor application) and produces the required term. Following the decorator pattern, a basic term factory may be wrapped by other term factories to realize alternative semantics, such as installing attachments for parent references or for origin tracking.

When executing a Stratego transformation, a single term factory is associated to the execution context of the Stratego runtime system. All term construction required by the transformation is delegated to this term factory. To enforce the application of typesmart constructors in place of regular constructors, we implemented a special term factory for typesmart term construction and changed the Stratego runtime system to use this term factory by default. In summary, the runtime support for typesmart constructors comprises the following components:

Typesmart term factory. A designated term factory for typesmart term construction that delegates term construction to typesmart constructors if available.

Term-sort attachment. Term metadata that indicates the sort of the term. This attachment is read and installed by typesmart constructors.

Typesmart primitives. Primitive functions for setting and retrieving term-sort attachments (`put-anno` and `get-anno` in our previous examples), and a primitive function for intentionally building an unsafe term. The latter primitive may only be used within typesmart constructors to build the resulting term after the arguments have been checked.

Has-sort library. The library described in Section 3.1, which is used within typesmart constructors.

Typesmart constructors. Language-specific typesmart constructors of the form described in Section 2. These may be automatically generated from an SDF syntax definition by our generator described in Section 4.

Caching typesmart term factory. A term factory which caches sorts of successfully constructed terms for later reuse without repeating validation.

When a user transformation requests the construction of a term $C(t_1, \ldots, t_n)$, this request is served by the augmented Stratego runtime system as follows.

1. The execution context dispatches the construction request to its term factory. Let us assume this term factory is a simple typesmart term factory without caching.
2. The typesmart term factory checks whether a typesmart constructor for constructor C with arity n exists. We represent typesmart constructors as regular Stratego transformations named `smart-` followed by the name of the constructor. Accordingly, we check for the existence of a Stratego transformation smart-C with arity n in the execution context.
   (a) If no transformation smart-C with arity n exists, then no typesmart factory has been defined for C. The typesmart term factory then calls a standard term factory that creates an unchecked term and returns it to the user's transformation. For example, this happens for lists, tuples, and optionals.
   (b) If a transformation smart-C with arity n exists, the typesmart term factory dispatches the construction of $C(t_1, \ldots, t_n)$ to this transformation as described in Section 3 with details as follows.
3. In the second case (b), the typesmart constructor checks the constructor arguments using a Stratego implementation of function `has-sort`. The subsort relation that `has-sort` uses is provided through additional transformation definitions in the execution context. Function `has-sort` uses a primitive function to retrieve the sort attachment of a term. If all arguments conform to the expected sorts, the typesmart constructor uses a primitive function to build an unchecked term through a standard term factory (this is needed to avoid cycles). Subsequently, the typesmart constructor applies a primitive function to install the result sort as a term-sort attachment. This term is returned.
4. A successful invocation of a typesmart constructor indicates a morphologically correct term and results in the term with installed term-sort attachment. A failed invocation of a typesmart constructor indicates a violation of the term signature and causes the execution of the transformation to be aborted.

In essence, the typesmart term factory only dispatches calls to typesmart constructors which are responsible for the actual verification and construction of terms.

Furthermore, we provide a term factory that caches the result sort of typesmart constructors to alleviate the runtime overhead of typesmart constructors. Our cache assumes that the result sort of a typesmart term construction only depends on the constructor name, the constructor's arity, and the sorts of the constructor arguments. This assumption holds for all typesmart constructors generated by our tool from an SDF grammar (Section 4). When the construction of a term is requested, the caching term factory checks whether a construction with the same constructor name, arity, and argument sorts occurred before. If not (cache miss), then the construction is delegated to the non-caching typesmart term factory. We cache the sort of the resulting term in an internal map. If a construction with the same constructor name, arity, and argument sorts occurred before (cache hit), we retrieve the result sort from the internal map, build an unchecked term, and install the cached result sort.

Thus, a cache hit avoids calling out to the Stratego transformation implementing the typesmart constructor, which furthermore avoids running function `has-sort` on all arguments.

Through the integration of typesmart constructors at the Stratego level, any technology building on top of Stratego enjoys dynamic enforcement of syntactic language constraints, too. Notably, Spoofax [16] and SugarJ [8] use Stratego as a metalanguage for the implementation of language analyses and semantics. We evaluate typesmart constructors and our implementation in Stratego through application in Spoofax and SugarJ.

6. Case studies

We evaluate the applicability and usefulness of the typesmart constructors by using them inside Spoofax [16] and SugarJ [7, 8]. 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.

We extended Spoofax such that it generates typesmart constructors from SDF definitions as explained in Section 4. This generation step is applied during compilation of a Spoofax language project. The typesmart constructors are compiled and loaded together with the user-supplied syntax, analysis rules, semantics, and editor services. We employ the instrumented Stratego runtime system described in Section 5 to transparently enforce syntactic validity during analysis, desugaring, and while executing semantic editor services.

We extended the SugarJ system to generate and use typesmart constructors for the base language. A SugarJ language extension extends the base language by defining additional syntax (as an SDF grammar), additional static analysis for the extended syntax (in Stratego), a desugaring from the extended syntax to the base language (in Stratego), and editor services for the extended syntax. We modified SugarJ to generate additional typesmart constructors for every user-defined extension. When an extension is activated via an import statement, we merge the typesmart constructors of the base language with the typesmart constructors of the extension. Again, we use the augmented runtime system of Stratego to transparently enforce syntactic validity.

To evaluate the applicability and usefulness of typesmart constructors, we applied them in Spoofax and SugarJ. In Spoofax, we applied typesmart constructors in a project that implements a compiler for a subset of Java. In SugarJ, we applied typesmart constructors for validating language extensions of Java and Haskell (transformations are implemented in Stratego). In addition, we evaluated the performance implications of typesmart constructors by measuring and comparing the execution time of a lambda-calculus compiler running with regular constructors only, with typesmart constructors, and with cached typesmart constructors.

6.1 MiniJava compiler

We evaluated the Spoofax integration of typesmart constructors by application to the Spoofax-based MiniJava implementation. The MiniJava compiler is a Stratego program that transforms MiniJava programs into their equivalent counterpart written in the Jasmin assembler language [1] for the Java Virtual Machine. The transformation translates a MiniJava syntax tree into Jasmin syntax tree, which is subsequently pretty-printed. As described above, we generated a library of Jasmin and MiniJava typesmart constructors from syntax definitions of Jasmin and MiniJava, respectively.

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

The MiniJava compiler was implemented, maintained, and thoroughly tested by a Stratego and Spoofax expert. Accordingly, we expected that the MiniJava compiler produces syntax trees that conform to the syntactic constraints of Jasmin. 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 morphologically incorrect Jasmin ASTs to be generated by the compiler. We describe the errors we found below.

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 [26] 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. For example, the trees `LabelRef(end)` and `end` are pretty-printed to the same string. Furthermore, we believe that the relatively low severity of the defects we found is due to compiler having been heavily tested prior to this evaluation. We believe that the high number of uncovered bugs to be indicative of the prevalence of bugs in other code generators that use abstract 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.

6.2 Language extensions of Haskell and Java

We applied typesmart constructors to language extensions developed with SugarHaskell [10] and SugarJ [8], which both are developed as instances of our framework for syntactic language extensibility [9]. This led to the discovery of a number of bugs in previously developed and tested language extensions. We describe the errors we found below.

In a SugarHaskell language extension that introduces special syntax for “idiomatic brackets”, we found a bug related to constructor and variable symbols in Haskell. At many places, the Haskell grammar distinguishes constructor symbols (starting with an upper-case character) from variable symbols (starting with a lower-case character). We failed to retain this distinction in our desugaring. We had to rewrite the production of the language extension and the desugaring to retain this distinction:

```
"(|" 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 the second change designates this symbol as a user-defined operator instead of a constructor in the generator. Would we break this distinction between constructors and variables (as the original code did), this may have far-reaching 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 probably fail. Our typesmart constructors notified us of the error immediately when the program was generated, instead of silently failing such that the error could have only been noticed when a generated and optimized program fails to perform as fast as expected.

Another two errors occurred in auxiliary functions that construct Haskell terms using fold functions. In the first case, we tried to construct a qualified module identifier from a list of strings using `foldr1` from the standard Stratego library. This failed because `foldr1` passes a singleton list containing the last list element to the argument functions. However, our function expected the last list element directly, and thus failed to unpack the singleton list. This led to the construction of the ill-formed term `QModId("Control", ["Applicative"])`. The revised version correctly generates `QModId("Control", "Applicative")`.

```
foldr1(id, \(x,y) -> QModId(x,y)\)
   ⇝ foldr1(\[x] -> x\, \(x,y) -> QModId(x,y)\)
```

In the second case, we tried to construct a pair-wise sequence of toplevel declarations from a list of declarations. This failed because we used `foldr` instead of `foldr1` to prevent the empty list ending up as a declaration.

```
foldr(id,\(x,y) -> TopdeclSeq(x,y)\)
   ⇝ foldr1(\[x] -> x\,\(x,y) -> TopdeclSeq(x,y)\)
```

Finally, similar as in the MiniJava compiler, we found a few instances of missing constructor applications that were supposed to wrap expression literals, variable symbols in expressions, etc.

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 generators indeed are safe and produce well-formed programs.

6.3 Performance benchmarks

We evaluated the performance penalty introduced by typesmart constructors. We benchmarked the performance of a compiler from lambda expressions to Java similar to the one in Figure 1. We compared the execution time of the generator for three lambda expressions of increasing size in four scenarios: (*nocheck*) no typesmart constructors, (*check*) non-cached typesmart constructors, (*cache*) cached typesmart constructors reset prior to execution, (*persist*) cached typesmart constructors with cache preservation across transformations. We repeated each execution three times and averaged the results, thus 36 total executions were performed. Table 1 summarizes the timings and cache statistics observed.

	S			M			L		
	T	H	M	T	H	M	T	H	M
nocheck	0.023	-	-	0.026	-	-	0.033	-	-
check	1.493	-	-	2.153	-	-	8.170	-	-
cache	1.207	30	32	1.167	63	32	1.193	405	35
persist	0.015	62	0	0.02	95	0	0.04	440	0

Table 1. Benchmarks results where S, M and L are a small, a medium, and a large input lambda expression; T, H and M are execution times in seconds, cache hits, and cache misses.

We observed that typesmart constructors without any form of caching (case *check*) introduce a large time penalty that is non-linearly related to the the size of the input program. The execution overhead is proportional to the number of terms constructed. The execution overhead also depends on the target language and more specifically on its syntax specification in SDF: The more alternative productions there are per sort, the higher the average number of checks a typesmart constructor has to perform. In our benchmark, we generate programs of the Java language, which has a rather large grammar of 1164 source lines of code.

Results from caching scenarios (*cache*) and (*persist*) show that term-sort caching as described in Section 5 is not only beneficial but necessary for performance. This necessity is clear from the approximately 50% cache hit to miss rate for even the smallest test case and up to 92% for the largest test case. The high cache hit ratios suggest two conclusions. Firstly, the overhead introduced by the typesmart constructors is significant for small transformations producing very heterogenous ASTs. For these short running transformations that cover many different constructors, the caching is unlikely to yield significant improvements. Secondly, larger transformations benefit from caching after the initial period required to fill up the cache. Furthermore, results from scenario (*persist*) confirm that the typesmart term factory does not induce any overhead by itself.

In future work, we want to investigate the application of *hybrid analysis* to reduce the runtime overhead of typesmart constructors. In many cases, typesmart constructors can be checked statically based on partial information. In particular, we can statically check syntax trees that occur literally in a transformation. Even in build patterns that integrate dynamically computed trees into a static skeleton, we can check the skeleton except for the immediate vicinity of the dynamic data. We expect that hybrid analysis can significantly reduce the runtime overhead of typesmart constructors while providing the same guarantees and supporting the same flexibility for transformation languages.

7. Discussion

In this section, we reflect on typesmart constructors and discuss support for potential problematic scenarios, additional application areas, and future extensions of the concepts presented in this paper.

7.1 Mixing typesmart and non-typesmart constructors

Typesmart rely on the assumption that (i) terms are built bottom-up and that (ii) subterms are built via typesmart constructors as well. While the first assumption is ubiquitous for the construction of tree-shaped data, the second assumption may fail to hold in some scenarios. For example, a legacy transformation libraries may employ internal, intermediate representations that are not accompanied by typesmart constructors. Multiple issues can arise when using such a legacy library together with a library that employs typesmart constructors.

First, if a non-typesmart term is used as a typesmart constructor argument, the argument is going to be rejected because it lacks a sort annotation. Most likely, this behavior is intended and correctly notifies the programmer that an unexpected term ended up as argument to the typesmart constructor. In some rare cases, however, it is possible that the unchecked argument in fact is well-formed and adheres to the expected sort. For example, this happens in Spoofax when a term is deserialized from an external ressource (e.g., loaded from a file). To ensure that the term indeed adheres to the sort requirement of the typesmart constructor, we can simply rebuild (that is, clone) it in the current execution using a typesmart term factory. If the term is well-formed, this must succeed and the term can afterwards be used like any other checked term.

Second, the legacy library might try to construct terms with typesmart constructors that the second library introduced. Since the legacy library was not aware of the additional constraints, it may attempt to temporarily create ill-formed terms. For example, some existing Stratego pretty-print libraries operate by incrementally stringifying a term bottom-up: `Plus(Plus(1, 2), 3)` transforms to `Plus("1+2", 3)`, which is illegal since the first argument of `Plus` is not of expression sort. For libraries that produce ill-formed intermediate terms but can be trusted to eventually provide well-formed result terms, we added a primitive second-order Stratego transformation `finally-typesmart(s)`. This transformation takes another transformation `s` as argument and executes it in an execution context with a standard term factory that does not perform any typesmart syntax checking. This allows us, for example, to call legacy pretty-print libraries that eventually yield a simple string. After transformation `s` yields a resulting tree, **finally**-typesmart checks the well-formedness of the generated tree by rebuilding (cloning) it as described in the previous paragraph.

Third, another problem occurs if the legacy library internally uses a constructor that is independently introduced as a typesmart constructor in the new library. To ensure composability of independently declared constructors, we introduced alternative sorts in Section 3: All equally-named constructors are represented by a single typesmart constructor that, given the actual argument terms, returns a term with all valid result sorts as alternatives. This pattern fails when two equally-named constructors exist but only one of them is typesmart. As described in Section 5, the typesmart term factory will execute the one typesmart constructor independent of which constructor was intended to be used. Thus, the typesmart constructor shadows the non-typesmart one. If these constructors do not incidentally expect the same argument sorts, the transformations of the legacy library are bound to fail. We have no clear solution for this scenario so far. One possibility might be to limit scope of typesmart constructors such that the legacy library is not included. We plan to further investigate the integration typesmart and non-typesmart transformations in our future work.

7.2 Typesmart constructors in OO languages

In this paper, we mainly explored the application of typesmart constructors in the transformation language Stratego. However, the ideas behind typesmart constructors can be applied in any language with a notion of construction. In particular, typesmart constructors can be used in object-oriented (OO) languages.

In dynamically typed OO languages such as Ruby, Python, or JavaScript, arguments of constructors are not checked at object-creation time at all, but only when a member of an argument is required. When encoding the abstract syntax of a programming language as a class hierarchy in a dynamically typed OO language,

```haskell
data Annotated a = Annotated { val :: a, freevars :: [String] }
put-freevars a vars = Annotated a vars

data Exp = Var String | Lam String AExp | App AExp AExp
type AExp = Annotated Exp

var :: String -> Annotated Exp
var s = put-freevars (Var s) [s]

lam :: String -> Annotated Exp -> Annotated Exp
lam s e = put-freevars (Lam s e) (delete s (freevars e))

app :: Annotated Exp -> Annotated Exp -> Annotated Exp
app e1 e2 = put-freevars (App e1 e2)
                         (freevars e1 ‘union‘ freevars e2)

finalize :: Annotated Exp -> Either Exp String
finalize e = case freevars e of
             [] -> Left (val e)
             xs -> Right ("Error: free variables " ++ show xs)
```

Figure 5. Typesmart constructors in Haskell that guarantee closed terms (no free variables).

we can use typesmart constructors to check the argument types at object-creation time. We can even allow the class hierarchy to deviate from the abstract syntax by again relying on annotations to store an object's sort (implemented as a member of the object). Moreover, the instrumentation of the Stratego runtime system (Section 5) can be mirrored for other languages to transparently apply typesmart constructors when available and to cache the checking of syntax sorts. This way, existing generators can remain unchanged and we can ensure the invariant that all objects that represent programs are well-formed with respect to the syntactic constraints of the language. This enables safer metaprogramming in dynamically typed OO languages. Furthermore, typesmart constructors can also be useful in statically typed OO languages if the class hierarchy is less precise than the syntactic constraints we want to enforce.

7.3 Mixing concrete and abstract syntax

As illustrated in Section 1, the use of concrete syntax in generation templates can preclude some well-formedness issues that arise when using abstract syntax. However, concrete syntax alone cannot guarantee the generation of syntactically well-formed code because spliced program fragments must be checked to match the expected sort.

In Stratego, a generation template that uses concrete syntax is preprocessed into a generation template that uses the corresponding abstract syntax [26]. Since the preprocessed template uses regular constructors, the modified Stratego runtime system would impose typesmart constructors also for those subterms where concrete syntax guarantees well-formedness. However, it is fairly straightforward to modify the preprocessor for concrete-syntax templates such that only spliced program fragments are checked, whereas all other subterms are built without run-time checking. Indeed, this resembles a hybrid analysis that can significantly improve run-time performance for transformations that employ lots of concrete syntax in generation templates.

7.4 Typesmart constructors for semantic constraints

Eventually, we want to use typesmart constructors not only to enforce syntactic constraints of a language, but also to enforce semantic constraints. We want to guarantee that only programs can be generated that adhere to semantic properties of a language such as name resolution or type checking. Syntactic constraints are relatively easy to enforce because they are *context-free*. This means that we can check whether a term is well-formed only by inspecting the term; we need no knowledge about how the term is used in different contexts. Semantic properties tend to be *context-sensitive* in nature. For example, for type checking we cannot give a final answer when only seeing a variable term. It depends on whether this variable is bound and used with the same type consistently.

Typically, semantic properties are enforced by traversing a term top-down. This way, it is possible to keep track of the context while going down. For example, a type checker keeps track of the bound variables and their types. When reaching a variable term, it suffices to look up the variable type in the accumulated context. However, we want to check programs while they are generated, and programs are generated bottom-up.

While the investigation of typesmart constructors for semantic constraints is not the focus of our present work, we discuss some early ideas here. First, since we cannot change the order in which terms are generated, we must adapt the checking of the semantic properties. Second, to make the order of checking semantic properties independent of the order in which knowledge about the program becomes available, we want to use a constraint system as explored before by Miao and Siek [19]. A constraint system has the advantage that additional constraints can be generated at any time. Nevertheless, whenever a new constraint is added to the constraint system, we can test whether the current problem is still satisfiable. This way, we hope to find ill-formed programs as soon as a constraint violation manifests, while allowing required knowledge for saturating constraints to emerge later on during the program generation.

For example, consider the Haskell program in Figure 5 that implements three typesmart constructors to ensure that generated lambda-calculus programs are closed terms, that is, they do not contain free variables. This is a simple, but context-sensitive property. Essentially, the construction of a term always yields the constructed term and a *residual constraint system* that has to be satisfied by the surrounding context. In our example, the residual constraints is simply a list of free variables, that have to be bound before the term conforms to the semantic property of being closed. Every occurrence of a variable adds a constraint. Every occurrence of a lambda saturates a constraint. As soon as a term has an empty list of free variables, it is well-formed. Otherwise, we never know whether additional context becomes available later on or whether the term is indeed ill-formed. To this end, we defined a function `finalize` that enforces the saturation of all constraints and reports an error otherwise. In our future work, we want to investigate whether typesmart constructors can be effectively used for enforcing semantic properties when generating programs.

8. Related Work

Using smart constructors to enforce data invariants has a long tradition and is a popular idiom in functional programming. The use smart constructors goes at least back to Stephen Adams, who used smart constructors to ensure tree balancing in the definition of an efficient set representation [1]. Our work on typesmart constructors goes considerably beyond traditional smart constructors: First, we allow typesmart constructors to annotate the constructed trees, which enables checking of data invariants without recursing into subtrees. Second, we integrate support for typesmart constructors into the run-time system to globally and transparently enforce invariants.

Smart constructors can also be used to implement local optimizations such as constant folding. For example, Elliott, Finne, and De Moore use optimizing smart constructors to compile the embedded language Pan [6]. In contrast, typesmart constructors must either fail or behave exactly as the corresponding regular constructor. Thus, typesmart constructors cannot be used to implemented optimizations directly. However, optimizing constructors call regular constructors after optimization has finished. By replacing these regular constructors with typesmart constructors (either manually or transparently by our run-time system), it is possible to combine the benefits of optimizing constructors and typesmart constructors.

Most related approaches for guaranteeing the syntactic or semantic well-formedness of generated programs significantly restrict the expressiveness of the metaprogramming language. For example, MetaML [22] supports type-safe run-time staging, MacroML [12] supports type-safe compile-time staging, and SoundExt [18] supports type-safe compile-time staging based on user-defined type rules. However, the expressiveness of the employed transformation languages is limited: MetaML and MacroML support no inspection of input terms; SoundExt supports inspection but requires small-step rewrite rules and cannot handle generic traversals. Language embeddings in form of generalized algebraic data types do not support flexible traversal and generation patterns as well as language composition.

To address metalanguages with sophisticated metalanguage features such as generic traversals or generating programs of composed languages, we explore the application dynamic analyses. In particular, typesmart constructors can be used for dynamic analysis in any metaprogramming system that has or can be retrofitted with a notion of term construction. Furthermore, typesmart constructors are independent of high-level language features such as generic traversals, because eventually all program generation is handled by term constructors.

Static checking of syntactic language constraints. The need for dynamic checking of term well-formedness is caused by the introduction of generic traversal strategies that do not fit in traditional static type systems. Traditional algebraic specification/rewrite languages such as OBJ [14], ASF [23], and later Maude [5] are statically typed. However, these languages have limited expressiveness that can be handled by type systems with first-order types extended with variations on subtyping such as injections and order-sorted algebra. For example, in ASF+SDF function signatures are defined with grammar productions, such as `"compile" "(" DslExp ")" -> JavaExp`. Rewrite rules (equations) are typechecked by parsing, avoiding the need for dynamic type checking. However, the first-order nature of these type systems does not admit the definition of generic, reusable transformation functions, leading to boilerplate code for e.g. traversals. Extensions such as ELAN's [3] congruence operators and ASF's traversal functions [24] fit within the first-order type system, but thereby limit the traversals that can be expressed.

Rascal [17] is the successor of ASF+SDF. Instead of defining transformation functions as extension of the object grammar, it provides a separate statically typed general-purpose language with domain-specific features to express metaprogramming. Rascal supports generic traversals via a built-in visitor pattern that allows the modification of the visited tree. Rascal's type system requires that the modified tree component has the same type as the original one. While this ensures that all generated trees are syntactically well-formed, it limits the flexibility of the transformation because it is not possible to transform a tree from one type to another.

The generalized (traversal) strategies of the Stratego [27] language provide an expressive and flexible transformation language, but is not covered by static type systems. This paper provides an approach to catch syntactic type errors in Stratego programs using a programmable dynamic type system.

Template engines. Template engines such as StringTemplate and Xpand generate code by directly composing strings. While these engines may guarantee type-safe input access, no checks are done on the output. Repleo [2] is a template engine that *does* provide syntactic guarantees about the output generated by templates. Partly this is based on the concrete object syntax approach also applied in ASF+SDF. This approach is extended by reparsing the result of strings that are substituted into the holes of a template. In contrast, the approach in this paper is based on tree rewritings, where the syntactic category of intermediate trees does not change and, thus, does not have to be rechecked. Repleo provides an interesting mix of static and dynamic analysis. Static analysis for parsing the template modulo splices, and dynamic analysis for checking the spliced strings. Our typesmart constructors can benefit from such a hybrid analysis as well, which could be achieved by partially evaluating the application of type smart constructors in statically known term fragments.

Stratego's concrete object syntax [26] provides partial static checking of syntactic templates in rewrite rules by parsing term fragments written in the concrete syntax of the object language. However, this technique fails to check the composition of term fragments. The dynamic approach in this paper checks all constructed terms and is complementary to concrete-syntax templates.

TemplateHaskell [20] is a template engine for Haskell. It provides syntactic safety via algebraic data types. Templates ensure scoping and type-safety statically up to spliced values, which are defined in a quotation monad that only ensures referential transparency statically and defers type checking until after the outermost splice has been computed.

Dynamic analysis. Miao and Siek explore dynamic type checking in the context of metaprogramming [19]. After every evaluation step of the metaprogramming language, they check the type of the generated code artifact by collecting and resolving constraints incrementally. Miao and Siek apply their technique in an idealized metaprogramming language [13] that resembles C++ metaprogramming and does not support materialization, introspection, or transformation of programs as data (e.g., abstract syntax trees cannot be manipulated directly). For this reason, the question of syntactic well-formedness does not even arise. In contrast, we explore dynamic checking of language-specific constraints in metaprogramming systems that permit arbitrary manipulation of code.

The `format-check` tool of Stratego/XT [4] checks terms against a signature by means of a full term traversal. This can be applied after a transformation or between stages of a transformation pipeline to ensure that a term conforms to the expected syntax. This approach can be used to catch errors in transformations. However, it can be hard to trace back a signature violation to an actual error in a transformation rule. Kalleberg and Visser [15] describe weaving of format checkers into Stratego programs as a case study of an experimental aspect-oriented extension of Stratego. A joinpoint on term construction can be used to inject the application of smart constructors as an alternative mechanism to our term factory replacement. In addition to the delivery mechanism, typesmart constructors apply to all intermediate terms and use annotations to avoid duplicate effort in analyzing the same subterms multiple times. Moreover, we mark a term with an alternative result sort if there is an overlap of signatures. This enables us to check and annotate terms despite the lack of context information.

Dynamic contracts as, for example, used in Racket [11, 21] also provide a programmable interface for run-time checking of structures. One important difference to our work is that contracts may not annotate the checked term. In contrast, we use annotations as a communication channel from subterms to parents. Moreover, contracts in Racket need to be manually programmed. We focus on dynamic analysis of metaprograms that generate other programs and provide a generator of typesmart constructors from a language's syntax definition.

9. Conclusion

We propose a novel approach for safe metaprogramming using dynamic analyses and typesmart constructors. A dynamic analysis is relatively easy to define since it can inspect run-time values. We explore the application of dynamic analyses for program generation, where the analysis is executed at program-construction time and can reject a partially generated program as soon as a violation manifests. In particular, we propose typesmart constructors as a mechanism to realize dynamic analyses for program generation. We demonstrate how to implement typesmart constructors for validating the syntactic well-formedness of generated programs and provide a tool to derive such typesmart constructors automatically from the syntactic definition of the target language. The caching of typesmart syntax checks ensures moderate run-time overhead and enables scalability to large generated programs. In our evaluation, we successfully applied typesmart constructors in Spoofax and SugarJ and found errors in preexisting program transformations.

Acknowledgments

We thank the anonymous reviewers for their helpful feedback.

References

[1] S. Adams. Functional pearls: Efficient sets–a balancing act. *Functional Programming*, 3(4):553–562, 1993.

[2] J. Arnoldus, J. Bijpost, and M. van den Brand. Repleo: A syntax-safe template engine. In *Proceedings of Conference on Generative Programming and Component Engineering (GPCE)*, pages 25–32. ACM, 2007.

[3] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of elan. *Electronic Notes in Theoretical Computer Science*, 15, 1998.

[4] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. *Stratego/XT Reference Manual*, 2003–2008.

[5] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of maude. *Electronic Notes in Theoretical Computer Science*, 4:65–89, 1996.

[6] C. Elliott, S. Finne, and O. De Moor. Compiling embedded languages. *Functional Programming*, 13(3):455–481, 2003.

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

[8] 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.

[9] 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.

[10] 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.

[11] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *Proceedings of International Conference on Functional Programming (ICFP)*, pages 48–59. ACM, 2002.

[12] S. Ganz, A. Sabry, and W. Taha. Macros as multi-stage computations: Type-safe, generative, binding macros in MacroML. In *Proceedings of International Conference on Functional Programming (ICFP)*. ACM, 2001.

[13] R. Garcia and A. Lumsdaine. Toward foundations for type-reflective metaprogramming. In *Proceedings of Conference on Generative Programming and Component Engineering (GPCE)*, pages 25–34. ACM, 2009.

[14] J. Goguen, C. Kirchner, H. Kirchner, A. Mégrelis, J. Meseguer, and T. Winkler. An introduction to OBJ 3. In *Conditional Term Rewriting Systems*, pages 258–263. Springer, 1988.

[15] K. T. Kalleberg and E. Visser. Combining aspect-oriented and strategic programming. *Electronic Notes in Theoretical Computer Science*, 147(1):5–30, 2006.

[16] 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.

[17] P. Klint, T. van der Storm, and J. Vinju. Rascal: A domain-specific language for source code analysis and manipulation. In *Proceedings of Conference on Source Code Analysis and Manipulation (SCAM)*, pages 168–177, 2009.

[18] F. Lorenzen and S. Erdweg. Modular and automated type-soundness verification for language extensions. In *Proceedings of International Conference on Functional Programming (ICFP)*, pages 331–342. ACM, 2013.

[19] W. Miao and J. G. Siek. Incremental type-checking for type-reflective metaprograms. In *Proceedings of Conference on Generative Programming and Component Engineering (GPCE)*, pages 167–176. ACM, 2010.

[20] T. Sheard and S. Peyton Jones. Template meta-programming for Haskell. In *Proceedings of Haskell Workshop*, pages 1–16. ACM, 2002.

[21] T. S. Strickland, S. Tobin-Hochstadt, R. B. Findler, and M. Flatt. Chaperones and impersonators: run-time support for reasonable interposition. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 943–962. ACM, 2012.

[22] W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2):211–242, 2000.

[23] M. van den Brand, A. van Deursen, J. Heering, H. De Jong, et al. The ASF+SDF Meta-Environment: A component-based language development environment. In *Proceedings of Conference on Compiler Construction (CC)*, volume 2027 of *LNCS*, pages 365–370. Springer, 2001.

[24] M. G. J. van den Brand, P. Klint, and J. J. Vinju. Term rewriting with traversal functions. *Transactions on Software Engineering Methodology (TOSEM)*, 12(2):152–190, 2003.

[25] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.

[26] 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.

[27] 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.