Statically Safe Program Generation with SafeGen

Shan Shan Huang, David Zook, and Yannis Smaragdakis

College of Computing, Georgia Institute of Technology,
Atlanta, GA 30332, USA
{ssh, dzook, yannis}@cc.gatech.edu

Abstract. SafeGen is a meta-programming language for writing statically safe generators of Java programs. If a program generator written in SafeGen passes the checks of the SafeGen compiler, then the generator will only generate well-formed Java programs, for any generator input. In other words, statically checking the generator guarantees the correctness of any generated program, with respect to static checks commonly performed by a conventional compiler (including type safety, existence of a superclass, etc.). To achieve this guarantee, SafeGen supports only language primitives for reflection over an existing well-formed Java program, primitives for creating program fragments, and a restricted set of constructs for iteration, conditional actions, and name generation. SafeGen's static checking algorithm is a combination of traditional type checking for Java, and a series of calls to a theorem prover to check the validity of first-order logical sentences constructed to represent well-formedness properties of the generated program under all inputs. The approach has worked quite well in our tests, providing proofs for correct generators or pointing out interesting bugs.

1 Introduction

Program generators can play an important role in automating software engineering tasks. A large amount of research has concentrated on meta-programming tools for writing program generators more conveniently or safely [4,5,15,7,13,11,2,6,3,17,18]. Nevertheless, such tools have not enjoyed much practical adoption. Programming language designers typically find meta-programming to be too unwieldy and undisciplined to be added as a general-purpose language feature. Working programmers who routinely use and write generators seem to find that advanced meta-programming infrastructure adds very little to what they can do with simple, text-based tools. For instance, many tens of thousands of programmers worldwide use code templates in the text-based XDoclet tool [12] to generate code for interfacing with J2EE application servers.

If a sophisticated meta-programming tool is to become mainstream, it should offer significant value-added for the generator programmer, comparable to the value added by high-level programming languages over assembly programming.

R. Glück and M. Lowry (Eds.): GPCE 2005, LNCS 3676, pp. 309–326, 2005.

In this paper, we explore one possible direction for adding such value. We present SafeGen: a meta-programming language that offers static guarantees on the correctness of the generator, yet is expressive enough for many practical applications. That is, a generator written in SafeGen is analyzed statically and its correctness is examined under *all* possible legal inputs, where the user specifies what constitutes a legal input. If the analysis succeeds, the generator is guaranteed to only produce well-formed Java code. This addresses a common problem in generator development and a major reason why meta-programming often appears too unwieldy and undisciplined: a generator may have bugs that cause it to produce illegal programs but only under certain inputs. Such bugs can stay undetected for a long time and may only be found by end users and not by the generator writer.

To achieve well-formedness guarantees, SafeGen has an easy-to-analyze language for describing generators. This offers restricted syntax for describing control flow, iteration, and name generation. Inputs to a SafeGen generator are limited to legal Java programs. That is, SafeGen generates programs by examining existing Java programs at a level comparable to that of Java reflection. All of SafeGen's reasoning is done in a logic that deals with reflective entities (e.g., methods of a class, argument types of a method, etc.), as opposed to, say, integer numbers. Intuitively, this makes SafeGen ideal for XDoclet-like [12] tasks. For instance, SafeGen is appropriate for going over an existing Java class and creating a delegator, or wrapper, or interface, or GUI class that will work correctly with the original class. In contrast, SafeGen is *not* appropriate for generation tasks such as creating specialized versions of the FFT transformation for specific matrix sizes and dimensions.

SafeGen statically checks the legality of code templates by combining traditional Java type checking algorithms with automated proofs of the validity of logical sentences. That is, SafeGen expresses the structure of the generator as a collection of first-order logic formulas, treated as axioms. Further axioms, also in first-order logic, encode standard properties of Java at the static checking level (e.g., the fact that a final class cannot be extended). Finally, correctness conditions of the generator are described as first-order logic conjectures. SafeGen uses an automated theorem prover, SPASS [16], to attempt to prove these correctness conditions under all inputs, based on the axioms.

SafeGen's contribution to the meta-programming research community is its novel approach of combining logic on reflexive properties of valid programs with program generation, to guarantee the legality of programs that are not generated until the run-time of the generator. This approach makes SafeGen the only meta-programming tool we know of that both guarantees at the compile time of the generator the type-correctness of the generated program, and allows generation of arbitrary pieces of code (potentially with references to free variables and unknown types). SafeGen also shows that despite the restriction on control flow and name generation, this approach still allows the expressiveness that is useful for many program generation needs. This general logic-based approach is not

limited to SafeGen's current target language, Java, but could be applied to other languages.

2 Motivation and Background

One can question whether static checking of a generator is a valuable feature. After all, once the generator is used, the generated program will be checked statically before it runs. So why try to catch these errors before the program is even generated? The answer is that static checking is not intended to detect errors in the generated program or even errors in the generator input, but errors in the generator itself. Although these errors will be detected at compile-time of the generated program, this is at least as late as the run-time of the generator. Thus, static legality checking for generators is analogous to static typing for regular programs. It is a desirable property because it increases confidence in the correctness of the generator under all inputs (and not just the inputs with which the generator was tested). To see the problem in an example, consider a program generator that emits programs depending on two input-related conditions: (We use MAJ [18] syntax: code inside a quote, `‘[...]`, is generated. The unquote operator, `#[...]`, is used to splice the result of an evaluated expression inside quoted code.)

```
if (pred1()) emit( ‘[int i;] );
...
if (pred2()) emit( ‘[i++;] );
```

If, for some input, `pred2` does not imply `pred1`, the generator can emit the reference to variable `i` without having generated the definition of `i`. This is an error in the generator. However, it might not surface until after the generator writer has tested and widely deployed the generator. This error will then be detected by some random user. It should be the responsibility of a good meta-programming language to prevent such errors by statically examining the generator.

The problem of guaranteeing the well-formedness of generated programs is essentially a problem of analyzing the control-flow and data-flow of the generator. For instance, in the above code fragment, the question is whether there is a valid program path that reaches the second `emit` statement without passing through the first. Similarly, consider a generator that introduces two new names in the same lexical context:

```
emit( ‘[ int #[name1], #[name2]; ] );
```

For static well-formedness checking, we need to know that `name1` and `name2` do not hold the same value (or we will end up with an illegal duplicate variable definition in the generated program). This is a data-flow property.

We should note that an interesting special case of program generation already offers strong legality guarantees for generated programs. Specifically, multi-stage languages, such as MetaML[13], MetaOCaml[7] or MetaD[10] guarantee that the generated program is type-correct by statically checking the generator. In this

sense, multi-stage languages represent the state of the art in static safety checking of generators. Nevertheless, staging applies restrictions on the structure of the generator and prohibits the expression of code templates in arbitrary fragments. Both of our above code examples are not possible in a multi-stage language. In the first example, identifiers in generated code (e.g., `i`) cannot refer to generated variable definitions that are not in an enclosing lexical scope inside the generator text. This is a drawback, even if the final program is expressible in a multi-stage language: ideally, a good meta-programming language should allow its user to express a generator in the style the user finds most convenient. In the second example, it is not possible in a multi-stage language to have the name of a generated definition vary depending on generator input.(Concretely, in MetaOCaml syntax, we cannot write, `.<let .~name:int = 0 in .~name + .~name)>.`, since binding instances cannot be escaped. Similarly, we cannot escape a type, e.g., `.<let i:.~typename = 0 in i+i>.`)

These restrictions mean that multi-stage languages are ideal for program specialization where the entire code to specialize is available, but not program generation where the generated program may be partial and may need to cooperate with other parts whose structure is not known until generator run-time. For example, a common generation task for J2EE applications is to take as input an arbitrary Java class and produce a Java interface that contains all of the class's public methods [14]. In this case, there is no code to specialize that is statically known to the generator. If the generator is to reason about the well-formedness of its output, it needs to do so using abstract properties of yet-unknown program entities, such as "no two methods in the input class can have the same type signatures". This is exactly the kind of program generation that SafeGen intends to support.[1] From a technical standpoint, the problem is harder than multi-stage programming, since there are no restrictions as to how the control and data-flow of the generator can influence the contents of the generated program parts.

3 SafeGen Design

In this section we describe the main design of the SafeGen language. We first give a high-level overview of SafeGen and then present the language in detail.

3.1 Overview of the Approach

Before we discuss the specifics of the SafeGen language, we will offer a quick example of what SafeGen can do, which will hopefully illuminate the role of all the distinct language features described in detail in the next sections. As we have not yet defined all the elements of SafeGen syntax and functionality, we will appeal to the reader's intuition for our example.

[1] We expect that the general approach used in SafeGen could also apply to program specialization tasks. Nevertheless, as mentioned earlier, SafeGen's current input language and reasoning engine is limited to reflection-like properties, and cannot apply to, say, generating specialized numerical code for a given array size and dimensions.

A basic, but not too interesting, SafeGen generator is the following:

```
#defgen makeInterface (Class c) {
  interface I {
    #foreach(Method m : MethodOf(m,c)) { void #[m] (); }
  }
}
```

The elements of this definition are as follows. The generator is called `makeInterface`. It accepts a Java class as its argument. It generates an interface named `I` (this name may be modified at generator runtime if the generator is used multiple times in the same lexical context). For each method of the input class, the generator produces a void, no-argument method by the same name in the generated interface.

Although this generator is almost trivial, it is still challenging to determine automatically whether it will output a valid interface for every input class. For example, do all the declared methods have unique signatures? In its attempt to prove that the generated code is well-formed, SafeGen relies on three kinds of knowledge: assumptions about the input (in this example there are none other than the fact that it is a class), general knowledge of Java typing, and the assumption that the input comes from a well-formed Java program (e.g., the input class, `c`, has methods with legal names). SafeGen uses the SPASS theorem prover to attempt to prove well-formedness properties of the output under any possible input. All knowledge that SafeGen has about the program is expressed as first-order logic sentences. For instance, the following formula states that any two members (either classes or interfaces) in a well-formed Java package need to have different names. (We show here the formula in SPASS syntax in order to be concrete about the level of interfacing with the theorem prover.)

```
formula(forall([c1, c2],
  implies(and(equal(DeclaringPackage(c1),DeclaringPackage(c2)),
              equal(Name(c1), Name(c2))),
          equal(c1,c2))),
         MEMBERS_IN_PACKAGE_DIFF_NAME).
```

The well-formedness conjectures that SafeGen tries to prove are also expressed as logic sentences. For instance, the following is a conjecture that states that generated methods cannot have the same name and type signatures if they are in the same class.

```
forall([m1, m2],
  implies(and(method(m1),
              method(m2),
              equal(DeclaringClass(m1), DeclaringClass(m2)),
              equal(Name(m1), Name(m2)),
              equal(Formals(m1), Formals(m2))),
          equal(m1, m2)))
```

In fact, this conjecture cannot be proven for the above generator, `makeInterface`. All generated methods have the same signature and methods can have the same names, since the same method name can be overloaded in the input class, `c`. Therefore, in this example we see that the output is potentially ill-formed.

3.2 Language Design

We describe next the syntax and main concepts of the SafeGen language. The language is described through short examples, followed by a longer example in the end. For a more thorough exposition, the syntax is shown in Figure 2 in the Appendix.

Cursors. The two main concepts in the SafeGen language are those of a *cursor* and a *generator*. A cursor is a variable ranging over all entities satisfying a first-order logic formula over the input program. Thus, the input program is viewed as a collection of logical facts about its type declarations. For instance, a cursor expression in SafeGen would be:

```
Method m : MethodOf(m,c) & Public(m) & !Abstract(m)
```

This cursor, `m`, describes all non-abstract, public methods in class `c` (`c` is a cursor assumed to have been defined earlier). In general, the values of cursors are type-system-level entities in the input program (methods, arguments, classes, interfaces, etc.). The logic predicates used to build cursors can be viewed best as a reflection mechanism over Java programs. SafeGen has several predefined predicates that correspond to Java reflection information and the user can create new predicate symbols that represent arbitrary first-order logic formulas over the predefined predicates. Since the logical sub-language used to define cursors in SafeGen is a standard first-order logic, we postpone describing its specifics in detail until later in the paper.

Generators. A SafeGen generator is a way to express Java code fragments. Generators are defined with the `#defgen` primitive. For example, a trivial generator, always producing a constant piece of code, is:

```
#defgen trivialGen () {
   class C { public void meth() {} }
}
```

A generator can receive input parameters that are either cursors or predicates describing constraints on the inputs. For instance, the following defines a generator that accepts a single non-abstract class as argument. (The body of the generator is elided.)

```
#defgen myGen (Class c : !Abstract(c)) { ... }
```

Similarly, one can define a new predicate that constrains the input of the generator:

```
#defgen myGen (input(Class c) => !Abstract(c)) { ... }
```

The above line defines a new predicate, called `input`, that is used to describe properties of the generator input values—namely that they are non-abstract classes. Note that (unlike predicate definitions that we will see later) the "implies" (`=>`) operator is used for predicates defining generator inputs: the input is not *all* classes that are non-abstract, just some classes that are guaranteed to be non-abstract.

A SafeGen generator interfaces with the outside world through Java reflection entities and strings. For instance, a generator that takes a `Class` argument, as above, is implemented as a Java method that accepts a `java.lang.Class` object as argument.

The body of a generator (enclosed in `{...}` delimiters) can contain any legal Java syntax. This Java code is "quoted"—that is, it gets generated when the generator executes. Quoted code can also contain three SafeGen constructs that serve as "escapes": they direct the control and data-flow of the generator, allowing configuration of the generated code. These three SafeGen constructs are `#[...]` (pronounced "unquote"), `#foreach` and `#when`.

The `#[...]` operator is used for adding fragments of Java code inside a larger fragment. For instance, a generator can integrate the output of another. More interestingly, a generator can derive code fragments by applying several built-in functions on cursors. Available functions are: `Name`, `Type`, `Formals`, `ArgNames`, `ArgTypes`, and `Modifiers`. Consider the example of the following generator:

```
#defgen myGen (Class c : !Abstract(c)) {
   #[c.Modifiers] class #[c.Name] {  }
}
```

This generates a new (empty) class with the same name and modifiers as the input class.

Functions `Name` and `Type` only generate one identifier, while `Formals` generates an array of $\langle ArgType, ArgName \rangle$ pairs, separated by commas. Functions `ArgNames`, `ArgTypes`, and `Modifiers` generate arrays of values, with `ArgNames`'s output separated by commas. Clearly, not all functions can be applied to all cursors. `Formals`, `ArgNames`, and `ArgTypes` can only be applied to `Method` cursors. SafeGen also allows the syntax `#[c]` on a cursor `c`. This is a shortcut for `#[c.Name]`.

The control flow of the generator is affected by primitives `#foreach` and `#when`, allowing iteration and conditional execution, respectively. SafeGen logic formulas determine iteration and conditional generation—thus, all iteration terminates and can only be over elements of the generator input.

The `#foreach` construct takes as argument a cursor definition. Inside the body of the `#foreach`, the cursor name can be used to refer to the current element in the range of the formula used to define the cursor. For instance, consider the following generator:

```
#defgen addFields (Class c) {
   #foreach ( Field f : FieldOf(f,c) ) { int #[f]; }
}
```

This creates a sequence of definitions of integer variables, each named after a field in the input class, `c`.

The `#when` construct's syntax is `#when ( LOGIC ) { CODE_TEMPLATE }`, optionally followed by `#else { CODE_TEMPLATE }`. That is, `#when` takes a logic formula as a parameter. If the formula evaluates to true at run-time, the first code template is generated. Otherwise, the code template following the `#else` is generated. In the example below, the argument to the generator is a set of Java interfaces (with no other constraints on them). If the set is not empty, then the "implements" clause gets generated, followed by all the names of interfaces. Otherwise, nothing gets generated.

```
#defgen maybeImplements ( input(Interface i) => true ) {
   #when ( exists (Interface in) : input(in) ) {
      implements #foreach(Interface i) { #[i] }
   }
}
```

Note that the above example also indicates that the generator's model ignores low-level separator tokens. I.e., our generators operate on abstract syntax trees, not parse trees. Thus, when the `#foreach` construct above generates multiple interface names, they get added to an AST. But when actual code is generated, they will be separated by commas, as Java requires.

User-Defined Predicates. For modularity and code reuse, SafeGen also allows definitions of new predicates both inside and outside the body of a generator. `#defpred` is used to give a name to a frequently used logic formula. The following example declares a predicate `myPred` that can be used in logic formulas, just like built-in predicates:

```
#defgen myGen ( ... ) {
   #defpred myPred ( Class c ) = Public(c) & !Final(c);  ...
}
```

Name Management and Hygiene. In the body of a generator, identifiers that correspond to generated definitions are hygienically renamed to avoid name conflicts. For instance, consider the following generator:

```
#defgen renameGen (input(Method m) => (m.Type = int) & noArg(m)) {
   #foreach( Method m: input(m) ) { int result = #[m](); }
}
```

(For convenience, the generator uses a predicate `noArg`, which we can define using `#defpred`. This constrains the input methods to accept no arguments.)

The result of the above generator will not be multiple definitions of variable `result`. Instead, at generation time, the actual variables generated will have

fresh names. Any references to these variables under the same cursor (or a cursor defined over a sub-range) will be consistently renamed to refer to the right variable. Since the renaming is only performed at the final output phase (i.e., when all generators have been called and the result is a complete Java compilation unit) SafeGen can tell which identifiers need renaming. Sometimes, a generator writer might indeed want to specify a name for a particular declaration, without renaming. In these cases, we provide the keyword `#name["..."]`. The identifier between quotes is generated as is.

Predicates, Cursors, and Logic in Detail. The logic underlying SafeGen is a sorted logic, with the basic sorts being: `Class`, `Interface`, `Method`, `Constructor`, `Field`, `Identifier`. Accordingly, all variables and constants in our domain are of one of these sorts. SafeGen does not provide any built-in constants. However, the user implicitly "creates" constants of the `Identifier` sort as needed. For example, if a user wishes to find all classes that implements `java.io.Serializable`, she writes the logical sentence:

```
forall (Class c) : (exists (Interface i) :
    ( InterfaceOf(i, c) & i.Name = "java.io.Serializable"))
```

`java.io.Serializable` is then declared as a constant in the domain during the compilation process.

The syntax for SafeGen logical sentences closely follows the syntax for first-order logic sentences (with the addition of sorts for declared variables). SafeGen provides logical operators `forall`, `exists`, `=`, `&`, `|` , `=>`, `!`, which correspond to all the operators available in first-order logic. The full list of available predicates and functions is shown in Figure 3 in the Appendix. For readers unfamiliar with first order logic syntax, please refer to Figure 2, rule `LOGIC` for details.

Example. We can now consider a non-trivial generator written in SafeGen. This is a realistic example, yet one that is short enough to study here and to use later for illustrating SafeGen's static checking process. The generator in Figure 1 takes a set of non-abstract classes as input and creates subclasses of the input classes with methods that just delegate to the superclasses' methods. (As explained earlier, the identifier `Delegator` is going to be renamed for each of the generated classes as to not induce name conflicts.)

3.3 Static Checking

We can now see how our approach can reason about a generator and guarantee that it produces well-formed programs under all inputs. Every well-formedness property of the output program is expressed as a logical formula. For instance, consider again our Section 2 example generator, for which we want to guarantee that a generated reference is always bound to a definition:

```
if (pred1()) emit( `[int i;] ); ... if (pred2()) emit( `[i++;] );
```

```
1. #defgen makeDelegator ( input(Class c) => !Abstract(c) ) {
2.    #foreach( Class c : input(c) ) {
3.       public class Delegator extends #[c] {
4.          #foreach(Method m : MethodOf(m, c) & !Private(m)) {
5.             #[m.Modifiers] #[m.Type] #[m] ( #[m.Formals] ) {
6.                return super.#[m](#[m.ArgNames]);
7.             }
8.          }
9.       }
10.   }
11. }
```

Fig. 1. A generator that generates a delegator class for an input class

The above example written in SafeGen is:

```
#when(logic_1) { int i; } ... #when(logic_2) { i++; }
```

where `logic_1` and `logic_2` are first-order logic formulas defined using built-in predicates and functions. Checking whether variable `i` is declared before use becomes checking the validity of the logical implication `logic_2` → `logic_1`. If the theorem prover proves validity, we know that under *any* input to the generator, the variable `i` would always be declared before it is used.

Other program well-formedness properties are also expressible in a similar fashion. Determining how to translate a given program property into a logical sentence is the role of the SafeGen implementation, described in the next section.

We should be explicit in that implementing checks for all well-formedness properties of Java programs is a heavy engineering task. SafeGen currently does not support all possible checks but we believe the omission is just a matter of engineering. [2] The currently supported checks in SafeGen are fairly representative in difficulty of the task and correspond to many valuable program correctness properties (e.g., method typechecking). Specifically, the currently fully supported tests are for the following properties.

- A declared super class exists.
- A declared super class is not `final`.
- Method argument types are valid.
- A returned value's type is compatible with the method return type.
- The return statement for a `void`-returning method has no argument.

[2] Any computable property can be expressed as the validity of a first-order logic formula. The only question is whether a theorem prover can reason about such properties effectively. For several yet-unsupported properties (i.e., properties for which SafeGen does not generate conjectures automatically) we have hand-produced logic formulas corresponding to example SafeGen programs and we have confirmed that we can reason about them in SPASS effectively. For instance, the conjecture in Section 3.1 was hand-produced, although our longer example in the Appendix (Figure 4) was automatically produced by the SafeGen compiler.

Notably missing checks include access control (e.g., no access to "private" variables outside class); checking for subtyping restrictions (e.g., a non-abstract class supplies definitions for all its superclass's abstract methods); checking for referring only to defined variables; checks for duplicate definitions; checking for correct declaration of exceptions; etc. We expect that many of them will be fully supported soon.

4 SafeGen Implementation

The most interesting part of the SafeGen implementation is the static checker. Therefore in this section we discuss how SafeGen produces axioms and proof obligations for a theorem prover, based on the structure of the SafeGen program.

4.1 SafeGen Static Checking

Although the SafeGen checking algorithm is not a traditional type-checker, it is easiest to present it in terms of type-checking, where both the names and the types of the various entities can depend on logic predicates.

SafeGen has two type-checking processes. One is type checking for the meta-language: legality of references to meta-variables, meta-level predicates, functions, and generators. (Meta-variables are either cursors or logic variables introduced by an `exists` or `forall` quantifier.) The second but much more complex one, is type checking for templated Java code. SafeGen's type system keeps two separate environments to support these two processes: the meta scope, for the generator, and the object scope, for the generated program.

Environment. A meta scope keeps track of meta level declarations: generators, predicates, and variables. A new meta scope is created by the following keywords: `#defgen`, `#defpred`, `#foreach`, `#when`, and quantifier keywords `forall` and `exists`. With the exception of `#when`, all of the keywords above create new meta-variable declarations. In addition to keeping track of declarations, `#foreach` and `#when` meta scopes are also associated with the logical sentences under which they are created. Each meta scope is linked to at most one parent meta scope. For example, in Figure 1, the meta scope created by `#foreach` on line 4 has the `#foreach` scope created on line 2 as a parent. The declarations in parent meta scopes are visible in the children scopes.

An object scope is much like a type environment for regular Java type checking. It contains symbol tables for types, variables, and methods. However, there are two unique elements of our object scope. First, all entries in the symbol table (e.g., names of variables or method declared in the scope, and the types these map to) may not be constants but dependent on a cursor over the input program. Second, each entry in the symbol tables has a link to a meta scope within which the entry is declared. For example, in Figure 1, class `Delegator`, declared on line 3, is an entry in the type table, with a link to the meta scope created on line 2, by `#foreach ( Class c : input(c))`. This meta scope in turn has a parent

meta scope corresponding to the `#defgen` in line 1. For an example of an object scope entry with an unknown name, consider the method declared on line 5 of Figure 1. The entry in the symbol table will contain the information that the method name is equal to the value of `m.Name` and the corresponding meta scope will be that defined by the `#foreach` on line 4 (with parent meta scopes those on lines 2 and 1). Only meta scopes created with `#defgen`, `#foreach`, `#when` can be linked from object scope entries.

Algorithm. SafeGen's type checking algorithm involves two phases. Phase I accomplishes the following two tasks:

1) Fully populate meta scopes and type check the meta language. Type checking the meta language is simply ensuring that a) every use of a meta-variable, predicate, function, or generator is defined, and b) if a meta variable is used as an argument to predicates, functions or generator calls, it has the correct type. For example, if meta-variables `m`, `c` are used in predicate `MethodOf(m, c)`, `m` should have a `Method` type, and `c` should have a `Class` or `Interface` type.

2) Collect type information in code templates. Object scopes are partially populated with only type information for declared types, their methods, fields, and inner types. No statements are inspected. There is no legality checking done in this phase. This step is analogous to a conventional type checking algorithm, where a first pass generates all the type information needed to type check the statements inside of method bodies and static initializers. After the object scopes are populated, we generate a logical representation of what is in the object scopes: a sentence describing the types available, their methods, fields, inner classes, etc. For the example in Figure 1, the initial segment of this sentence is:

```
forall([c],
  implies(and(Class(c), input(c)),
          exists([c'], and(Class(c'), Name(c')=Delegator, ...)))))
```

We call this sentence *fact*. It will be used in Phase II of the type checking algorithm, as described next.

Phase II is responsible for checking the type correctness of templated Java code. The algorithm resembles regular Java type checking in that it utilizes the symbol tables to look up information on variables, methods, and types. However, the algorithm is complicated by the use of meta-variables and functions in declarations and references. Therefore, SafeGen's type system combines the use of object scope symbol tables with the building of logical sentences using the meta scopes (i.e., the meta scope associated with the current object scope and all its parent meta scopes). For example, in Figure 1, we need to check whether the method call, `super.#[m](#[m.ArgNames])` on line 6 is a valid call. The first step is to look up the superclass of the current class using the symbol table. However, we find that `super` does not point to an actual class with its own symbol tables, but to a meta-variable, `#[c]`. In order to check whether `super.#[m](#[m.ArgNames])` is a valid call, we must construct a logical sentence to inquire: under all legal inputs to this generator (any class that is `!abstract`), and under the logical context

(encoded by the meta scope) in which this method call is used (namely, `#foreach(Class c:input(c)) {...#foreach(Method m:MethodOf(m,c))}`), does the class `#[c]` always have a method with name `#[m]`, and argument types the type of `#[m.ArgNames]`? This question is expressed as a logical sentence, *test*. The *test* sentence for the method call `super.#[m](#[m.ArgNames])` is shown in Figure 4 in the Appendix.

We then construct the sentence *fact* → *test*, where *fact* was constructed in Phase I, as described earlier. *fact* needs to be the condition in the implication because it states the existence of classes and methods that *test* might refer to. Facts about the well-formedness of generator inputs are also part of the theorem prover input, supplied as axioms. We next feed this sentence to the theorem prover to test its validity. The full input to the theorem prover includes the logic definition (i.e., predicates, functions, sorts), axioms about Java, and the *fact* → *test* conjecture. This is typically many hundreds of lines long.

5 Discussion

Using the Theorem Prover. There are two approaches to using the theorem prover to verify the correctness properties of code templates. We could construct a large sentence that is the conjunction of all the type-correctness properties the templated code should preserve, and ask the prover whether these properties hold given the facts produced by the code templates. While this approach simplifies our language implementation by delegating all type checking duties to the theorem prover, it has a major disadvantage. The checking would be all-or-nothing and it would not produce very useful error messages to the users. When one of the properties in the conjunction fails to be valid due to a contradiction, all we receive from the theorem prover is a series of syntactic maneuvers that arrived at the contradiction. It is very difficult to decipher these messages to determine the exact property that failed. We can only inform the user that *somewhere* in their program, there is an error. The problem is exacerbated by spurious errors due to valid formulas that could not be proven: the user would be unable to tell that the error is spurious if we just reject the entire program.

Therefore, we have chosen a second approach. SafeGen's type checking algorithm is a combination of traditional Java type checking and calls to the theorem prover. We make calls to the theorem prover to check the validity of very specific properties. For example, when we are type-checking a class declaration, and we reach the declaration of a super class, we make two calls to the theorem prover. One is to check that the declared super class exists. Another is to check that the super class is a non-final class. This approach yields simpler logic formulas to prove. At the same time, we are able to produce very precise error messages to the user regarding exactly which property the code template failed to establish.

The one disadvantage of our approach is that we must make many calls to the theorem prover in the process of compiling just one generator. There might be a potential performance hit depending on how long the theorem prover takes to return answers. However, as discussed next, we have not yet found this to be a major cause of concern.

Experience. SafeGen is still work in progress. Nevertheless, we have experimented extensively with the checking process for formulas that correspond to SafeGen programs. In fact, we first chose example SafeGen programs and expressed in logic their properties that we wanted to check, before trying different theorem provers and eventually choosing SPASS.

The choice of theorem prover is largely orthogonal to the overall approach, and we may switch in the future. The overriding factor we used in choosing a theorem prover was its ability to arrive at a result without human guidance. We cannot expect the user of SafeGen to hand-tune the logic whenever the theorem prover fails. A theorem prover that fails to find either a definite proof of validity or a counterexample would cause SafeGen to produce lots of spurious warnings to users. After trying several (4) theorem provers, we chose SPASS because (in our tests) it demonstrated the best ability to terminate much of the time without human guidance. With our limited set of example validity tests, SPASS always finds a proof for the valid sentences. For sentences that are not valid, SPASS terminates with a decision roughly 50% of the time. It fails to terminate (during the several minutes we observed it) the other 50% of the time. This means that, for our examples, SafeGen issues no false positive errors. However, for half of the true type errors SafeGen reported, SafeGen was only able to report a "possible error", because SPASS did not terminate with a decision (i.e., a counterexample) that the sentence is not valid.

Because SafeGen makes a large number of calls to the theorem prover during type-checking, the performance of the theorem prover was a consideration, as well. So far, for the cases that SPASS was able to terminate, it terminates in under 1 second. This is hardly surprising: most of the properties we want to prove are quite shallow. For instance, for many type-checking tests, the types and meta scopes match exactly even though they are complex expressions involving cursors and logic predicates. Currently we set the time limit for each SPASS proof attempt at 3 seconds.

It is worth noting that our delegator example in Figure 1 has a bug that SafeGen readily detects: the superclass method is not always guaranteed to have a return type. If the return type of method `m`, called in line 6, is `void`, then the statement `return super.#[m](#[m.ArgNames])` is not legal. The user should instead use a `#when` clause, to detect whether the superclass method has a returnable result and if not to just call it without attempting to return its value.

Another result of our experiments with properties of sample SafeGen generators is that we tuned our logic to limit its expressiveness but maximize the number of proofs we can produce completely automatically. That is, when we find in our examples that a specific pattern causes consistent difficulties in reasoning, we remove the logic feature it depends on. For instance, transitivity is very hard to reason about. The superclass relation is transitive, but instead of specifying the transitivity fully in our logic axioms, we only expand it three levels. As a result, if the validity of a generator depends on a subtyping relation between classes more than 3 links away in the subtyping hierarchy, then our logic cannot express the proof and SafeGen will issue a spurious warning.

Big Picture: Soundness and Why a New Language? The SafeGen static checking algorithm is sound: if a generator is approved by SafeGen, it is guaranteed to be correct (with respect to the supported tests, of course—but with no fundamental reason why these tests cannot be all possible Java well-formedness tests). As in any static checking system, however, what matters most is not soundness but usefulness. After all, soundness is easy to achieve by just rejecting all programs. In the static checking arena, tools like ESC/Java [8] have garnered a lot of attention by trying to be useful, even though they are not sound.

We view the soundness argument as tied to another major decision, namely whether to support a hard-to-analyze programming language like Java as the meta-language, or to design a small, specialized language like SafeGen. If we were to implement our checking approach on a meta-programming system built on top of Java (such as our MAJ system [18]), we would certainly have sacrificed soundness to achieve usefulness. Java has several language constructs (including dynamic dispatch, aliasing and assignments, exceptions) that make it hard to be sound (i.e., guarantee correctness) while allowing a large percentage of the correct programs. Instead, our choice of creating a new language was largely so that we could be sound, yet useful. We believe that soundness is not a goal by itself, yet it is valuable in terms of user perception. Sound static checking mechanisms (such as type systems) are much more easily accepted by programmers than unsound tools (like `lint` or ESC/Java) because they feel more disciplined. At the same time, we have aimed at making SafeGen expressive enough for most program generation tasks that depend on reflection over existing programs.

Of course, SafeGen checking offers no guarantees of completeness: if we find no proof of the correctness of the generator, it is by no means certain that it is erroneous. Since first-order logic is undecidable, the proof process will not always terminate. We have examined the possibility of restricting our language to a broad but decidable fragment of first-order logic, such as the guarded fragment [1]. (In fact, SPASS, with the right choice of parameters is a decision procedure for the guarded fragment [9].) Nevertheless, we believe that this would limit significantly the expressiveness of our logic. Furthermore, it is not clear whether a guarantee of termination of the proof process with a decision is a very important property in practice, unless it is a guarantee of termination in a very short time, which seems impossible: such decision procedures typically have super-exponential complexity.

6 Conclusions

In this paper we presented SafeGen, a meta-programming language with the distinguishing feature that it offers powerful correctness guarantees for generators expressed in it. SafeGen statically checks its input to guarantee that only well-formed code will be generated at the generator's runtime. We demonstrated a novel approach that combines traditional static type checking with representing program correctness properties in logic. We believe that SafeGen is expressive and useful, even though its syntax is restricted so we can represent all program

correctness properties logically. We also believe that the approach of using logic to control and reason about code generation is one that extends beyond the implementation of SafeGen. It can be used for a different target language (from Java), and with a different logic (from one based on Java reflexive properties), suitable for other broad categories of generation needs.

Acknowledgments. This research was supported by the National Science Foundation under Grants No. CCR-0220248 and CCR-0238289.

References

1. H. Andreka, J. van Benthem, and I. Nemeti. Modal languages and bounded fragments of predicate logic. *Journal of Philosophical Logic*, 27(3):217–274, 1998.
2. J. Bachrach and K. Playford. The Java syntactic extender (JSE). In *Proceedings of the OOPSLA '01 conference on Object Oriented Programming Systems Languages and Applications*, pages 31–42. ACM Press, 2001.
3. J. Baker and W. C. Hsieh. Maya: multiple-dispatch syntax extension in Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 270–281. ACM Press, 2002.
4. D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: tools for implementing domain-specific languages. In *Proceedings Fifth International Conference on Software Reuse*, pages 143–153, Victoria, BC, Canada, 1998. IEEE.
5. D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. In *Proceedings of the 25th International Conference on Software Engineering*, pages 187–197. IEEE Computer Society, 2003.
6. A. Bryant, A. Catton, K. De Volder, and G. C. Murphy. Explicit programming. In *Proceedings of the 1st international conference on Aspect-Oriented Software Development*, pages 10–18. ACM Press, 2002.
7. C. Calcagno, W. Taha, L. Huang, and X. Leroy. Implementing multi-stage languages using ASTs, gensym, and reflection. In *Generative Programming and Component Engineering (GPCE) Conference*, LNCS 2830, pages 57–76. Springer, 2003.
8. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245. ACM Press, June 2002.
9. H. Ganzinger and H. de Nivelle. A superposition decision procedure for the guarded fragment with equality. In *Logic in Computer Science conference (LICS)*, pages 295–304, 1999.
10. E. Pasalic, W. Taha, and T. Sheard. Tagless staged interpreters for typed languages. In *International Conference on Functional Programming (ICFP)*, pages 218–229, New York, NY, USA, 2002. ACM Press.
11. T. Sheard and S. P. Jones. Template meta-programming for Haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 1–16. ACM Press, 2002.
12. A. Stevens et al. *XDoclet Web site, http://xdoclet.sourceforge.net/.*
13. W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Partial Evaluation and Semantics-Based Program Manipulation, Amsterdam, The Netherlands, June 1997*, pages 203–217. New York: ACM, 1997.

14. E. Tilevich, S. Urbanski, Y. Smaragdakis, and M. Fleury. Aspectizing server-side distribution. In *Proceedings of the Automated Software Engineering (ASE) Conference*. IEEE Press, October 2003.
15. E. Visser. Meta-programming with concrete object syntax. In *Generative Programming and Component Engineering (GPCE) Conference*, LNCS 2487, pages 299–315. Springer, 2002.
16. C. Weidenbach. SPASS: Combining superposition, sorts and splitting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier, 1999.
17. D. Weise and R. F. Crew. Programmable syntax macros. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 156–165, 1993.
18. D. Zook, S. S. Huang, and Y. Smaragdakis. Generating AspectJ programs with Meta-AspectJ. In *Generative Programming and Component Engineering (GPCE) Conference*, volume 3286 of *Lecture Notes in Computer Science*. Springer, Oct. 2004.

Appendix

```
GENERATOR_DEF : "#defgen IDENT (" ( INPUT )? ") {" CODE_TEMPLATE "}" ;
INPUT         : CURSOR_DEC ("," INPUT )*
              | INPUT_PRED_DEC ("," INPUT )*  ;
CODE_TEMPLATE : JAVACODE
              | "#foreach (" CURSOR_DEC ") {" CODE_TEMPLATE "}"
              | "#when (" LOGIC ") {" CODE_TEMPLATE "}"
                ( "#else {" CODE_TEMPLATE "}" )?
              | GENERATOR_DEF
              | PRED_DEF  ;
CURSOR_DEC    : METATYPE IDENT ( ":" LOGIC )? ;
METATYPE      : "Class" | "Interface" | "Method" | "Constructor" | "Field";
INPUT_PRED_DEC: IDENT "(" ( PRED_ARGS )? ") =>" LOGIC ;
PRED_DEF      : "#defpred IDENT "(" ( PRED_ARGS )? ") =" LOGIC ;
PRED_ARGS     : METATYPE IDENT ("," METATYPE IDENT)* ;
JAVACODE      : Java syntax + "#[" + METAEXPR + "]" ;
METAEXPR      : IDENT ( "." META_FUN )*
              | IDENT "(" ( GEN_ARGS )? ")" ;
GEN_ARGS      : IDENT ( "," IDENT )* ;
META_FUN      : "Name" | "Type" | "Formals" | "ArgTypes" | "ArgNames" ;
LOGIC         : "forall" METATYPE IDENT : "(" LOGIC ")"
              | "exists" METATYPE IDENT : "(" LOGIC ")"
              | IDENT "=" IDENT
              | "!" LOGIC  |  LOGIC "&" LOGIC  |  LOGIC "|" LOGIC
              | LOGIC "=>" LOGIC ;
```

Fig. 2. SafeGen syntax

- Unary predicates: `Public`, `Private`, `Protected`, `Static`, `Final`, `Abstract`, `Transient`, `Strinctfp`, `Synchronized`, `Volatile`, `Native`
- Binary predicates: `PackageOf`, `ClassOf`, `InnerClassOf`, `InterfaceOf`, `SuperClassOf`, `ConstructorOf`, `MethodOf`, `FieldOf`, `ExceptionOf`, `ArgTypeOf`
- Functions: `Name`, `Type`, `Formals`, `ArgNames`, `ArgTypes`, and `Modifiers`.

Fig. 3. Available predicates and functions in SafeGen

```
implies(
forall([c],
  implies(
   and(input(c), class(c)),
   exists([del],
     and(class(del),
         equal(Name(del), Delegator),
         forall([sc], equiv(equal(SuperClass(del), sc), equal(c, sc))),
         forall([m],
           implies(and(equal(DeclaringClass(m), c), method(m)),
                   exists([del_meth],
                     and(method(del_meth),
                         equal(DeclaringClass(del_meth), del),
                         equal(Name(m), Name(del_meth)),
                         equal(RetType(m), RetType(del_meth)),
                         equal(Formals(m), Formals(del_meth)))))))))),
forall([c],
  implies(and(input(c), class(c)),
  forall([m],
    implies(
     and(equal(DeclaringClass(m), c), method(m)),
     exists([meth],
       and(method(meth), equal(Name(meth), Name(m)),
           exists([sc],
             and(equal(DeclaringClass(meth), sc),
                 exists([c'],
                   and(equal(DeclaringClass(meth), c'),
                       equal(SuperClass(c'), sc),
                       equal(Name(sc'), Delegator))))),
         equal(Formals(meth), Formals(m)))))))))
```

Fig. 4. SPASS Conjecture for type-validity of "super" call in example