A Language Generic Solution for Name Binding Preservation in Refactorings

Maartje de Jonge
Delft University of Technology
Delft, Netherlands
M.deJonge@tudelft.nl

Eelco Visser
Delft University of Technology
Delft, Netherlands
E.visser@tudelft.nl

ABSTRACT

The implementation of refactorings for new languages requires considerable effort from the language developer. We aim at reducing that effort by using language generic techniques. This paper focuses on behavior preservation, in particular the preservation of static name bindings. To detect name binding violations, we implement a technique that reuses the name analysis defined in the compiler front end. Some languages offer the possibility to access variables using qualified names. As a refinement to violation detection, we show that name analysis can be defined as a reusable traversal strategy that can be applied to restore name bindings by creating qualified names. These techniques offer an efficient and reliable solution; the semantics of the language is implemented only once, with the compiler being the single source of truth. We evaluate our approach by implementing a language generic rename refactoring, which we apply to two domain specific languages and a subset of the Java language.

Categories and Subject Descriptors

D.3 [**Programming Languages**]: Miscellaneous

General Terms

Languages

Keywords

Static Analysis, Refactoring

1. INTRODUCTION

The successful development of new languages is currently hindered by the high cost of tool building. Developers are accustomed to the integrated development environments (IDEs) that exist for general purpose languages, and expect the same services for new languages. For the development of Domain Specific Languages (DSLs) this requirement is a particular problem, since these languages are often developed with fewer resources than general purpose languages.

Language workbenches aim at reducing that effort by facilitating efficient development of IDE support for software languages [18]. Notable examples include MontiCore [27], EMFText [22], MPS [25], TMF [7], Xtext [14], and our own Spoofax [26]. The Spoofax language workbench generates a complete implementation of an editor plugin with common syntactic services based on the syntax definition of a language in SDF [35]. Services that require semantic analysis and/or transformation are implemented in the Stratego transformation language [12]. Especially challenging is the implementation of refactorings, which are transformations that improve the internal structure of a program while preserving its behavior. Even with specialized compiler techniques such as rewriting languages and attribute grammars, implementing conditions for behavior preservation is a tough task that requires global understanding of the semantics of the language for which the refactoring is developed.

Traditionally, conditions for behavior preservation are implemented as preconditions that are checked before the transformation [31, 32]. This approach has some clear weaknesses: It is extremely difficult to derive a correct set of preconditions that guarantees behavior preservation without excluding refactorings that in fact could be carried out. Moreover, additional preconditions have to be implemented in case the language evolves. Finally, preconditions are not easily shared among different refactorings, nor do they transfer to different languages. Current refactoring implementations rarely guarantee behavior preservation. Rather, refactorings are tried on examples and validated 'in the field', which may result in subtle errors triggered by corner cases not foreseen by the developer. Even mature refactoring frameworks used in current IDEs contain bugs as a result of insufficient preconditions [33, 34].

The limitations of preconditions are addressed in [16, 33]. The authors propose an invariant based approach which is implemented in JastAdd [15], an attribute grammar system that extends Java with support for circular reference attribute grammars (RAGs) [21]. Invariants for name binding, control-flow and data-flow are implemented as complementary analysis functions that are checked after the transformation. Compared to refactoring frameworks used in existing IDEs, Java refactorings implemented in JastAdd proved to be more reliable and required less effort in terms of lines of code. The success of this approach can be explained by the fact that, compared to preconditions, the specification of the invariants more closely follows compiler analysis that

LDTA '12 Tallinn, Estonia

define the static semantics of a language.

Attribute grammars allow for a high-level declarative specification of semantic analysis, however, they offer no specific language features to declaratively express syntax tree transformations. An alternative approach is to use term rewriting for implementing refactorings. Term rewriting is used in systems such as Maude [19], Tom [9], Strafunski [28] and Stratego [12]. Term rewriting makes it easy to describe syntax tree transformations, but is less declarative with respect to semantic analysis. Rewriting systems typically view syntax trees as terms without a concept of node identity; AST nodes are characterized only by their subtrees, and not by their position within the whole tree. Reference attributes crucially rely on node identity, and hence have no direct equivalent in a term-based representation of syntax trees. In this paper we implement an invariant based approach within the paradigm of term rewriting.

The paper focuses on preservation of name bindings. All refactorings that introduce new names into a scope have to guard against accidental changes of existing name bindings, which change the semantic behavior of the program. We implement a preservation criterion for statically known name bindings that is generic applicable to different languages and different refactorings. The preservation criterion takes as arguments a language specific name analysis and a refactoring specific transformation.

Many languages offer the possibility to access variables defined in a namespace via qualified names. A notable example is Java; a field that is shadowed by a local variable can still be accessed using a qualifier such as `this` or `super`. As a refinement to the preservation criterion, we show that name analysis can be implemented as a reusable traversal strategy that can be applied to restore name bindings by creating qualified names, thereby enabling refactorings that would otherwise be rejected.

The described techniques for name binding preservation and name binding restoring form the main contribution of the paper. Contrary to prior approaches, these techniques adhere to the "Single Source of Truth" principle [5], with the compiler being the single, authoritative representation of semantic behavior. To see how our approach works in practice, we implemented a language generic rename refactoring which we applied to Stratego [12], MiniJava [8] and Mobl [24]. Our experience shows that the effort it takes to implement the rename refactorings is significantly reduced by using language generic techniques. Furthermore, manual and automated testing demonstrate that the implemented refactorings indeed preserve semantic behavior.

2. MOTIVATION

Refactorings are behavior preserving source-to-source transformations with the objective of improving the design of existing code [17]. Although it is possible to refactor manually, tool support reduces evolution costs by automating error-prone and tedious tasks. In particular, behavior preservation is automatically checked; in case the transformation changes the semantic behavior of the program, the refactoring is rejected and the problem is reported to the user.

Name bindings associate identifiers with program entities such as variables, methods and types. Name bindings form a semantic concern that should be preserved by refactorings. Intuitively, all name accesses in a program should bind to the same declarations before and after the transformation. The problem of maintaining existing name bindings occurs in many refactorings, with renaming being the obvious example. Automatically renaming an identifier requires binding information to determine which names must be renamed. Furthermore, behavior preservation requires that conflicting and accidentally shadowed declarations are detected.

Shadowing occurs when the same identifier is used for different entities in nested lexical scopes. Shadowing poses a challenge for binding preservation in refactorings, Figure 1 illustrates variable shadowing. The `age` variable in the body of the `setAge` method refers to the `age` field of the surrounding class. After renaming `s` to `age` (Figure 1, lower left), the age field is shadowed by the method parameter `age`, declared in an inner scope. As a consequence, the binding of the `age` variable has changed. As a minimal requirement, refactoring tools are expected to detect name binding violations to guarantee the save application of the refactoring.

A namespace is a scope that groups related identifiers, and allows the disambiguation of homonym identifiers residing in different namespaces. In many programming languages, identifiers that appear in namespaces have a short (local) name and a long "qualified" name. By using the qualified name, an identifier can still be accessed in an inner scope, even though a program entity with the same name is declared within that inner scope. Figure 1 (lower right) provides an example. By adding a name qualifier (`this`) to the `age` variable, the original name bindings are restored so the refactoring can still be carried out. Thus, the policy to reject all refactorings with name binding violations is in fact too restrictive, preventing refactorings that only require a small modification to be applied. Therefore, a demanded feature for refactoring tools is that they are able to restore violated name bindings by creating qualified names.

Name binding analysis is implemented by the compiler of a language, involving complex semantic rules for namespaces, scoping and visibility. Name binding preservation conditions in refactoring frameworks must implement the same semantic rules. This can hardly be guaranteed with an ad hoc, precondition based approach. Indeed, existing refactoring implementations based on preconditions contain bugs because of (new) language features that are not supported [33]. A more reliable solution can be realized when the criterion for name binding preservation is directly based on the name analysis implemented in the compiler. This way, the semantics assumed by the refactoring tool is guaranteed to be consistent with the semantics implemented in the compiler, even when the language evolves. We implement a name binding preservation criterion that reuses the existing name analysis defined in the compiler front-end. As a refinement, we present an approach for restoring name bindings. Using strategic rewriting, we show that name analysis can be defined as a reusable traversal strategy which is used to analyze name bindings as well as to check and restore name bindings by creating qualified names.

This paper is organized as follows. The name binding preservation criterion is discussed in Section 3, while Section 4 shows how name analysis can be reused to check and restore name bindings. Section 5 reports on our experience with applying both techniques to different languages.

3. PRESERVING NAME BINDINGS

Refactorings must preserve the name binding structure of a program, implicitly defined by the name analysis im-

```
class Person {
  int age;
  void setAge(int s){
    age = s;
  }
}
```

```
class Person {
  int age;
  void setAge(int age){
    age = age;
  }
}
```

```
class Person {
  int age;
  void setAge(int age){
    this.age = age;
  }
}
```

Figure 1: Renaming `s` to `age`, incorrectly and correctly applied to code fragment.

```
ClassDec(
  "Person"{"n0"}
, [ FieldDec("int", "age"{"n1"})
  , MethodDec(
      Void(), "setAge"{"n2"}, [Param("int", "age"{"n3"})]
    , [Assign(QA(This(), "age"{"n1"}), Var("age"{"n3"}))]
)])
```

Figure 2: Name annotations make bindings explicit. The reference names 'n1' and 'n3' distinguish the parameter 'age' from the field 'age'

plemented in the compiler. To make the binding structure explicit, we use the technique of explicit renaming implemented in Stratego by means of term annotations [11]. The result of the name analysis is an abstract syntax tree in which all identifiers are annotated with a globally unique reference name. That is, two identifiers are annotated with the same reference name if and only if they bind to the same declaration. Figure 2 shows the abstract syntax tree of the Java fragment of Figure 1 (lower right) in the ATerm format [10]. The names in the AST are annotated with unique reference names. The reference names 'n1' and 'n3' distinguish the parameter 'age' from the field 'age'. This information can be used for example in a rename refactoring to determine which identifiers must be renamed.

We use name annotations to implement name binding preservation as a post condition on the transformed tree. The preservation condition compares the existing annotations, which represent the original bindings, with the new annotations, obtained by reanalyzing the transformed tree. Preservation is realized if and only if the existing name annotations are equal to the new name annotations, modulo renaming. The criterion does not make any assumptions about the reference names that are generated for the declarations, except that they are unique. In particular, existing reference names may change after re-analysis. Furthermore, the criterion applies to refactorings that change the AST structure, as long as the name annotation in the transformed AST represent the original binding structure.

Figure 3 illustrates the preservation condition for our running example. For convenience, only the relevant name annotations are shown. The lower right and upper code fragments are equivalent modulo renaming, that is, the name annotations follow the same pattern (`[a,b,a,b]`). By contrast, the name annotations in the lower left fragment follow a different pattern (`[a,b,b,b]`), which means that the fragment has a different binding structure.

Figure 4, `apply-refactoring` implements the name binding violation detection, taking a language specific name analysis strategy and a refactoring specific transformation rule

```
int age{n1};
void setAge(int s{n3})
{
  age{n1} = s{n3};
}
```

```
int age{o1};
void setAge(int age{o3})
{
  age{o3} = age{o3};
}
```

```
int age{p1};
void setAge(int age{p3})
{
  this.age{p1} = age{p3};
}
```

Figure 3: Name binding pattern `[n1, n3, n1, n3]` in the upper fragment, has binding violations in the lower left fragment `[o1, o3, o3, o3]`, and is preserved in the lower right fragment `[p1, p3, p1, p3]`.

```
apply-refactoring(transform, analyze, is-name):
  ast -> (ast-t', violations)
  where
    ast-t            := <analyze; transform> ast;
    ast-t'           := <analyze> ast-t;
    old-names        := <collect-all(is-name)> ast-t;
    new-names        := <collect-all(is-name)> ast-t';
    violations       := <zip; binding-violations>
                            (old-names, new-names)

binding-violations:
  [(x,y)|tl] -> <conc>
      (hd-violations, <binding-violations <+ ?[]> tl')
  where
    hd-violations := <filter(is-binding-viol(|x,y))> tl;
    tl'           := <filter(not(?(_,y) <+ ?(x,_)))> tl

is-binding-viol(|x,y) =
  (?(_,y) <+ ?(x,_)); not(?(x,y))
```

Figure 4: `binding-violations` collects binding violations by comparing the reference names in the transformed tree, before and after re-analysis. [a].

[a]The syntax $< r >$ is used for rule application in Stratego, $s1; s2$ represents the sequential operator, first apply $s1$, then apply $s2$ to the resulting term

as arguments. The refactoring application returns the tree that results after applying the transformation, plus a (possible empty) list of name binding violations. The name binding violations are collected by comparing the original name annotations (`old-names`) with the new name annotations (`new-names`), in the order in which they occur in the transformed tree. All name tuples that break the implicit mapping between old and new names are returned as binding violations by the `binding-violations` rule. To improve performance, the `is-name` pattern can be implemented refactoring specific to enforce that only the endangered names are checked.

4. CREATING QUALIFIED NAMES

In the previous section we implemented a language generic preservation criterion that is parameterized with an existing name analysis strategy. In this section we propose an alternate approach. Instead of passing the name analysis as a parameter to the preservation criterion, we pass the preservation criterion as a parameter to the name analysis. The advantage of this approach is that we can extend the binding preservation condition with a binding repair rule. We show how a reusable name analysis traversal can be applied to set name binding annotations (Section 4.2), to detect name binding violations (Section 4.3), and to restore name bind-

```
class A { int a; }
class B extends A { int b; }
class C extends B {
  int c;
  class D extends E {
    int d; int x;
    void foo() {
      int i;
      i = <name>; }}}
class E { int e; int x;}
```

Figure 5: Name lookup in Java proceeds in an outward-upward motion.

ings by creating appropriate qualified names (Section 4.4).

Name lookup requires contextual information such as the set of visible declarations, nesting and inheritance relations between namespaces, and type information. The main task of the name analysis strategy is to propagate this contextual information to the access terms where a lookup rule is applied that returns the associated declaration. Using the Stratego paradigm of reusable traversal strategies, we assume that the rule applied at the access terms is passed as a parameter to the name analysis traversal. Since we treat name analysis as a black box, we will not provide implementation details. Example implementations of name analysis in Stratego are given in [23] and [26].

As a leading example, we implement our approach for a subset of Java that models some essential features of the Java language with respect to resolving name bindings. Supported features are: class inheritance and class nesting, local variable, field and method declarations, and variable access by simple or qualified names. Most other Java features are excluded.

4.1 Name Lookup in Java

Figure 5 shows a small Java fragment which features class inheritance, class nesting and variable declarations. To illustrate name lookup in Java, we show how name bindings are resolved for variable accesses at the `<name>` location. The declaration is looked up in the set of visible declarations, starting at the most local scope and proceeding in an "outwards-upwards" motion. In the given example, the declared variables are looked up in the following order: local declarations (`i`), fields declared in the enclosing class (`d` and `x`), fields declared in the super class (`e` and `x`), the outer class (`c`), and the super classes of the outer class (`b` and `a`). The field `x` in the super class `E` is not accessible by its simple name `x`, since it is shadowed by the field `x` in class `D`. However, the shadowed field can still be accessed by its qualified name, `super.x`. Qualifiers force the lookup to start at a specific namespace, thereby skipping over the names declared in nested or inner scopes.

4.2 Setting Name Binding Annotations

Figure 6 shows the Stratego code for name lookup in Java. The lookup logic is implemented in the `lookup` strategy; which progresses outwards lexically (`ns-out`), taking a detour upwards the inheritance hierarchy (`ns-up`). Lookup starts at the local namespace for local variables (`Var(name)`), and at the namespace associated to the qualifier for qualified access (`QA(qualifier, name)`). When the processed namespace contains a declaration with the given name, the `Reference` rule applies successfully and the reference name is returned. Otherwise, name lookup proceeds with the next,

```
annotate-names:
  ast -> <analyze-names(annotate-name)> ast

annotate-name:
  Var(name) -> Var(name{ref-name})
  where
    ref-name := <lookup> (Local(), name{})

annotate-name:
  QA(qualifier, name) -> QA(qualifier, name{ref-name})
  where
    ref-name := <lookup> (<get-type> qualifier, name{})

get-type : This() -> <CurrentClass>          // this
get-type : Super() -> <CurrentClass; Super> // super
get-type : QThis(ctype) -> ctype             // C.this
get-type : QSuper(ctype) -> <Super> ctype    // C.super
get-type : CastRef(ctype, t) -> ctype        // ((A)t)

lookup:
  (ns, name) -> ref-name
  where
    <while-not(
      while-not(
        // looks up reference for name
        // in namespace (Local() or classname)
        ref-name := <Reference> (<id>, name),
        ns-up
      ),
      ns-out
    )> ns

// lookup rules for the name of the
// super class, current class and outer class
ns-up : cl -> <Super> cl
ns-out: Local() -> <CurrentClass>
ns-out: cl -> <Outer> cl
```

Figure 6: `annotate-names` sets reference annotations at variable accesses. The reference is looked up in an outward-upward motion.

more global namespace set by `ns-up` or `ns-out`. `Reference`, `CurrentClass`, `Super` and `Outer` are dynamic rules, the Stratego equivalent of symbol tables. Unlike standard rewrite rules in Stratego, dynamic rules are created at run-time, and propagate information available at their creation contexts.

The `lookup` strategy is called by the `annotate-name` rule which sets the looked up reference name as annotation to the access term. The name annotation rule is passed as a parameter to the name analysis strategy `analyze-names`, which takes care of propagating the required context sensitive information to the access terms by creating dynamic rules. The name annotation rule is implemented in the compiler front-end as complement of the name analysis traversal.

4.3 Checking Name Bindings

Name binding violations can be detected by passing a violation check rule to the name analysis, offering an alternative to Section 3. The bindings that held before the transformation are stored in the tree as annotations, the bindings after the transformation are looked up during the re-applied name analysis (`collect-binding-violations`, Figure 7). To allow the comparison of old and new bindings, we assume that declarations keep their reference name when revisited. The comparison of the old and new binding is implemented by the `is-binding-violation` rule (Figure 7), which succeeds if original and newly looked up reference annotations are different. In case a binding violation is detected, a binding violation error is stored (`BindingViolation`) which will

```
collect-binding-violations:
  ast -> violations
  where
    <analyse-names(store-binding-violation)> ast;
    violations := <bagof-BindingViolation>

store-binding-violation:
  access -> access
  where
    <is-binding-violation> access;
    rules(BindingViolation:+ access)

is-binding-violation:
  access -> access
  where
    new-annotated-access := <annotate-name> access;
    <not(equal)> (access, new-annotated-access)
```

Figure 7: Binding violations are collected by comparing the old- and newly analyzed binding annotations during a name analysis traversal.

later be reported to the user.

4.4 Restoring Name Bindings

A name binding violation occurs when the original reference of an access term is shadowed by a name declared in an inner scope. The name binding is restored when a qualified name forces the lookup in a more general namespace. The code fragment in Figure 8 implements name binding repair for the Java (sub)language. The `create-qualified-access` rule restores the name bindings of violated access terms. The rule repeatedly creates qualified names that enforce lookup in a more general namespace, until a qualified name is constructed that preserves the original binding. The preservation is checked by applying the `is-binding-violation` rule discussed before. As with the name lookup strategy (Figure 6), the "outwards upwards" motion is followed. The `cast-up` and `cast-out` rules implement the construction of a new qualified name that targets the next, more general namespace. The walk across namespaces requires contextual information which is propagated during the name analysis traversal `analyze-names` implemented in the compiler front end.

5. EXPERIENCE

In this section we report on our experience with implementing refactorings for different languages, namely Mobl, Stratego and the subset of Java discussed in the paper. In the case of Mobl and Stratego, we used the existing compilers [3, 6], for the Java subset we implemented the compiler from scratch.

Genericity. To see if our techniques are applicable to different languages, we implemented rename refactorings for the Java, Mobl and Stratego compilers. The implementations show that the preservation condition is applicable to languages with statically known name bindings. To realize behavior preservation for Java we had to cope with method overriding and hence dynamic dispatch. Changes introduced in method overriding may change the behavior of the program without changing (static) name bindings. We implemented an additional semantic condition that checks if a refactoring changes the overriding structure of the program in which case a warning is reported. The technique is not applicable to dynamic languages were the name bindings are not statically known.

```
restore-name-bindings:
  ast -> <analyze-names(restore-binding)> ast

restore-binding:
  access -> qualified-access
  where
    qualified-access := <
      create-qualified-access <+
      rules(BindingViolation:+ access)
    > access

create-qualified-access:
  access -> qualified-access
  where
    qualified-access :=
      <while-not(
        while-not(
          where(not(is-binding-violation)),
          cast-up
        ),
        cast-out
      )> access

//cast-up and cast-out qualified names
cast-up:  QA(qualifier, name) ->
          QA(<cast-up-qualifier> qualifier, name)
cast-out: QA(qualifier, name) ->
          QA(<cast-out-qualifier> qualifier, name)
cast-out: Var(name) -> QA(This(), name)

//cast-up and cast-out qualifiers
cast-up-qualifier:  This() -> Super()
cast-up-qualifier:
  Super() -> CastRef(<get-type; ns-up> Super(), This())
cast-up-qualifier:
  CastRef(c, v) -> CastRef(<ns-up> c, v)
cast-up-qualifier:  QThis(c) -> QSuper(c)
cast-up-qualifier:
  qs@QSuper(t) -> CastRef(<get-type; ns-up>qs, QThis(t))
cast-out-qualifier:
  This() -> QThis(<get-type; ns-out> This())
cast-out-qualifier:
  QThis(t) -> QThis(<get-type; ns-out> t)
```

Figure 8: `restore-binding` restores a violated name binding by creating a qualified name. The qualified name is created by repeatedly up(out)-casting the qualifier until the binding preservation check succeeds. [a]

[a] $s1 <+ s2$ is Stratego syntax for guarded left choice. First $s1$ is applied, and only if it fails, $s2$ is applied to the original term.

Applicability. In this paragraph we comment on the effort it took to realize the implementations for the mentioned languages. Name binding preservation for Mobl and Stratego was accurately checked by applying the preservation criterion of Section 3. The clear benefit of using this technique was that no extra code had to be written to implement behavior preservation. However, the existing name analysis had to be tailored before it could be applied to check name binding preservation. The existing analysis was implemented on single files, resolving imported files from the file system. However, our approach requires that all affected ASTs are (re)analyzed in memory, since the correctness of the refactoring is evaluated in memory, before the files in the file system are modified. We coped with this issue by slightly modifying the existing name analysis so that it takes into account a list of in-memory ASTs when resolving im-

ports.

For Java (subset) we applied the technique discussed in Section 4.4 which offers support for restoring name bindings by creating qualified names. Some language specific code had to be written to implement the creation of qualifiers, the code is shown in Figure 8. Implementing the name analysis as a parameterized strategy hardly took any additional effort since it fits well in the Stratego paradigm.

Correctness. To evaluate the correctness of our approach, we manually tested rename refactorings on existing projects in Mobl and Stratego. In addition, we implemented test suites that cover critical cases for Mobl, Stratego and the Java sub-language. The test results confirm the correctness of the preservation criterion. Finally, we implemented an automated test strategy that uses an inverse oracle [13] to test whether the name bindings are preserved. First, a list of potential harmful names is created by collecting all names that appear in the program. Then, rename refactorings are applied to all names in the program with the new name randomly chosen from the set of potential harmful names. The renaming is only applied in case no binding violations are detected. As a last step, rename refactorings are applied that revert the names of all declarations to their original name. The inverse oracle states that the resulting tree is equal to the original tree (modulo annotations). We successfully applied this automated test strategy to evaluate the correctness of the preservation criterion of Section 3 on the Stratego compiler front end (written in Stratego). During the application of the test strategy, 484 renamings were applied, while 196 renamings were rejected because of binding violations.

Performance. The final important question to evaluate is if the implementation is practical with respect to performance. To address this issue, we performed a number of renamings on the source code of the Mobl compiler, which consists of about 8000 lines of Stratego code. Our experience shows that the approach scales at least to this size of projects. We did not yet do performance tests on larger programs.

6. RELATED WORK

This paper presents a language generic approach to name binding preservation, reusing the existing name analysis defined in the compiler of the language. We describe a preservation condition which is directly applicable to different languages and different refactoring transformations. Furthermore, we show that name analysis can be defined as a reusable traversal strategy that is applied to set binding annotations, to check bindings and to restore bindings by creating qualified names. These techniques offer an efficient and reliable solution; efficient because the refactoring developer does not have to implement complex conditions for name binding preservation, reliable because the semantics of the language is implemented only once, with the compiler being the single source of truth. Our implementation uses the technique of strategic term rewriting, and is implemented in Stratego.

Precondition Approaches. Behavior preservation of refactorings has been a primary concern in refactoring research. Opdyke [31] and Roberts [32] propose a precondition based approach. Preconditions specify which conditions a program has to meet for the refactoring to be correct. Limitations of a precondition based approach are pointed out in [33]. Complex scope nesting rules common in current languages make it hard to define sufficient preconditions, furthermore, additional preconditions have to be implemented in case the language evolves. A second limitation is that preconditions are often too strong, preventing refactorings that could be carried out with some small modifications. The mentioned paper gives examples where widely used refactoring tools (Eclipse [1], NetBeans [4], and IntelliJ [2]) admitted unsound rename refactorings where names did not bind to the correct declarations after the renaming.

JastAdd. JastAdd [15] implements an approach based on reference attribute grammars [21] that allow to express name analysis in a concise and modular manner. The JastAdd approach to name binding preservation [33] is based on the idea of inverted lookup functions. The inverted lookup functions compute names for variable accesses that bind to a given declaration. The access computation can be tailored to create qualified names, allowing the refactoring to proceed where otherwise a conflict would occur. After a refactoring transformation is performed, all endangered accesses are updated so that they resolve to the same declaration as before. If the updating fails, the refactoring is rejected and all changes are undone. The inverted lookup functions closely follow the lookup functions that specify the name analysis, as noticed in the paper, the size of the access computation code compares to the size of the lookup code. The correspondence between lookup and access computation helps to avoid many pitfalls overseen in precondition based approaches and to adjust the access computation when new language features are introduced. However, the process of inverting can not be automated. As a consequence, the invertible look up functions introduce duplication and therefore vulnerabilities for subtle bugs.

Formal Specification. In [19], the authors present a formal approach to the specification and verification of refactorings. Refactorings are specified formally, with conditional rewrite rules in the form of executable Maude equations. The refactoring specifications extend the equational semantics of the language at hand. Given a formal specification of the Java semantics, they provide detailed correctness proofs for behavior preservation of two Java refactorings. The approach can be used in conjunction with any language for which an equational semantics has been provided.

Refactoring Tools for Functional Languages. Huiqing Li et al. [30] present HaRe, a refactoring framework for Haskell. The HaRe framework makes use of the static analysis provided by the Haskell compiler frontend Programatica [20]. Transformations and analysis are implemented using Strafunski [29], a library for functional strategic programming in Haskell. Instead of an invariant based approach, the behavior preservation is implemented with help of pre- and postconditions and possible compensation strategies in case the conditions are violated.

Lammel [28] sketches the idea of a generic refactoring framework that could be instantiated for a variety of languages. The implementation is based on functional strategic programming in Haskell [29], and includes generic transformations and analysis functions as building blocks. The

generic functions are parameterized with the language-specific ingredients. The intention of the paper is to investigate the idea of a generic refactoring framework, but its applicability does not seem to go beyond a proof of concept.

7. CONCLUSION

We are developing a language generic refactoring framework with the objective to reduce the effort it takes to implement refactorings for new languages. By reusing the existing compiler infrastructure, we implemented a language generic solution to name binding preservation. We successfully applied the binding preservation criterion to implement renaming for different languages, which resulted in reliable implementations that required only a small effort in terms of lines of code. As future work, we plan to implement additional preservation conditions for control- and data- flow. Furthermore, we intend to implement a library of language generic refactorings that are directly applicable to new languages.

8. REFERENCES

[1] Eclipse. `http://www.eclipse.org`.

[2] IntelliJ. `http://www.jetbrains.com/idea`.

[3] Mobl compiler. `https://github.com/eelcovisser/mobl`.

[4] NetBeans. `http://netbeans.org`.

[5] Single Source of Truth. `http://en.wikipedia.org/wiki/Single_Source_of_Truth`.

[6] Stratego IDE. `https://svn.strategoxt.org/repos/StrategoXT/spoofax-imp/trunk/org.strategoxt.imp.editors.stratego`.

[7] Textual modeling framework. `http://www.eclipse.org/modeling/tmf`.

[8] The MiniJava Project. `http://www.cambridge.org/us/features/052182060X`.

[9] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. Tom: Piggybacking rewriting on java. In *RTA'07*, pages 36–47, 2007.

[10] M. G. J. van den Brand, H. de Jong, P. Klint, and P. Olivier. Efficient annotated terms. *Software, Practice & Experience*, 30(3):259–291, 2000.

[11] M. Bravenboer. Term Annotation. `http://www.program-transformation.org/Stratego/TermAnnotation`.

[12] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, 2008.

[13] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In I. Crnkovic and A. Bertolino, editors, *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 185–194. ACM, 2007.

[14] S. Efftinge and M. Voelter. oAW xText: A framework for textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit*, 2006.

[15] T. Ekman and G. Hedin. The jastadd extensible java compiler. In R. P. Gabriel, D. F. Bacon, C. V. Lopes, and G. L. S. Jr., editors, *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, pages 1–18. ACM, 2007.

[16] T. Ekman, M. Schäfer, and M. Verbaere. Refactoring is not (yet) about transformation. In *Proceedings of the 2nd Workshop on Refactoring Tools*, WRT '08, pages 5:1–5:4, New York, 2008. ACM.

[17] M. Fowler. Refactoring: Improving the design of existing code. volume 2418 of *Lecture Notes in Computer Science*, page 256. Springer, 2002.

[18] M. Fowler. Language workbenches: The killer-app for domain specific languages?, 2005.

[19] A. Garrido and J. Meseguer. Formal specification and verification of java refactorings. *Source Code Analysis and Manipulation, IEEE International Workshop*, 2006.

[20] T. Hallgren, J. Hook, M. P. Jones, and R. B. Kieburtz. An overview of the programatica toolset. In *High Confidence Software and Systems Conference, HCSS04, http://www.cse.ogi.edu/ hallgren/ Programatica/HCSS04*, 2004.

[21] G. Hedin. Reference attributed grammars. *Informatica (Slovenia)*, 24(3), 2000.

[22] F. Heidenreich, J. Johannes, S. Karol, M. Seifert, and C. Wende. Derivation and refinement of textual syntax for models. In *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, ECMDA-FA '09, pages 114–129, Berlin, Heidelberg, 2009. Springer-Verlag.

[23] Z. Hemel, L. C. L. Kats, and E. Visser. Code generation by model transformation. In A. Vallecillo, J. Gray, and A. Pierantonio, editors, *Theory and Practice of Model Transformations, ICMT*, volume 5063 of *Lecture Notes in Computer Science*, pages 183–198. Springer, 2008.

[24] Z. Hemel and E. Visser. Declaratively programming the mobile web with mobl. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011*. ACM, 2011.

[25] JetBrains. Meta programming system. `https://www.jetbrains.com/mps`.

[26] L. C. L. Kats and E. Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In W. R. Cook, S. Clarke, and M. C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, pages 444–463. ACM, 2010.

[27] H. Krahn, B. Rumpe, and S. Völkel. Monticore: Modular development of textual domain specific languages. volume 11 of *Lecture Notes in Business Information Processing*, pages 297–315. Springer, 2008.

[28] R. Lämmel. Towards generic refactoring. *Workshop on Rule-Based Programming*, cs.PL/0203001, 2002. informal publication.

[29] R. Lämmel and J. Visser. A strafunski application letter. In *Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*, PADL '03, pages 357–375, London, UK, UK, 2003. Springer-Verlag.

[30] H. Li and S. J. Thompson. Tool support for

refactoring functional programs. In R. GlÃijck and O. de Moor, editors, *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM 2008, San Francisco, California, USA, January 7-8, 2008*, pages 199–203. ACM, 2008.

[31] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, 1992.

[32] D. B. Roberts. Practical analysis for refactoring. Technical report, Champaign, IL, USA, 1999.

[33] M. Schäfer, T. Ekman, and O. de Moor. Sound and extensible renaming for java. In G. E. Harris, editor, *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, pages 277–294. ACM, 2008.

[34] G. L. Soares. Making program refactoring safer. In J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, editors, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010*, pages 521–522. ACM, 2010.

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