Towards language-parametric semantic editor services based on declarative type system specifications

Pelsmaeker, Daniel A.A.; van Antwerpen, Hendrik; Visser, Eelco

DOI
10.4230/LIPIcs.ECOOP.2019.26

Publication date
2019

Document Version
Final published version

Published in
33rd European Conference on Object-Oriented Programming, ECOOP 2019

Citation (APA)
Pelsmaeker, D. A. A., van Antwerpen, H., & Visser, E. (2019). Towards language-parametric semantic editor services based on declarative type system specifications. In A. F. Donaldson (Ed.), *33rd European Conference on Object-Oriented Programming, ECOOP 2019* (Vol. 134). Article 26 Schloss Dagstuhl-Leibniz-Zentrum fur Informatik GmbH, Dagstuhl Publishing. https://doi.org/10.4230/LIPIcs.ECOOP.2019.26

Towards Language-Parametric Semantic Editor Services Based on Declarative Type System Specifications

Daniel A. A. Pelsmaeker
Delft University of Technology, Delft, The Netherlands
d.a.a.pelsmaeker@tudelft.nl

Hendrik van Antwerpen
Delft University of Technology, Delft, The Netherlands
h.vanantwerpen@tudelft.nl

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

Abstract

Editor services assist programmers to more effectively write and comprehend code. Implementing editor services correctly is not trivial. This paper focuses on the specification of semantic editor services, those that use the semantic model of a program. The specification of refactorings is a common subject of study, but many other semantic editor services have received little attention. We propose a language-parametric approach to the definition of semantic editor services, using a declarative specification of the static semantics of the programming language, and constraint solving. Editor services are specified as constraint problems, and language specifications are used to ensure correctness. We describe our approach for the following semantic editor services: reference resolution, find usages, goto subclasses, code completion, and the extract definition refactoring. We do this in the context of Statix, a constraint language for the specification of type systems. We investigate the specification of editor services in terms of Statix constraints, and the requirements these impose on a suitable solver.

2012 ACM Subject Classification Software and its engineering → Semantics

Keywords and phrases semantics, constraint solving, Statix, name binding, editor services, reference resolution, code completion, refactoring

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.26

Category Brave New Idea Paper

Funding This research was partially funded by the NWO VICI *Language Designer's Workbench* project (639.023.206).

Acknowledgements We thank the anonymous reviewers for their feedback on previous versions of this paper. We also thank Arjen Rouvoet for his comments and his work on Ministatix, an implementation of the core Statix language we use for prototyping.

1 Introduction

Editor services, such as syntax highlighting, reference navigation, and variable renaming, are an important tool for programmers. For example, code navigation is important for effective comprehension of code [13], and refactoring approaches rely heavily on good tool support [8]. It is therefore no surprise that such services are regularly used by users of IDEs [9].

Editor services can be classified into syntactic and semantic editor services. The former, such as syntax highlighting, rely only on the (abstract) syntax of a program. The latter depend on a semantic model of the program, and use type or name binding information. Semantic editor services can be further divided in two groups: services that *inform* about the program, and services that *transform* the program. Informing services depend on the program model that is the result of type checking. The program model contains information on the types of variables, the declarations that references refer to, etc. Transforming services are guided by the program model (e.g., to rename a declaration and all its usages), but may also rely on the typing rules to ensure the resulting, transformed program is well-formed.

Implementing semantic editor services and ensuring their correctness is not a trivial task (see, e.g., the difficulties around correctly implementing Java refactorings [14]). Language workbenches are tools to aid the development of programming languages and programming environments [5] by means of declarative formalisms and reusable tools that support correctness and reduce development effort. A good example of this is the use of a context-free grammar to specify syntax. This specification can be used to drive a parser, but also for unparsing, or to provide syntactic code completion. The language developer writes a declarative specification, which helps with the correctness of the syntax, while existing parsing, unparsing, and code completion algorithms can be reused, reducing development time.

However, even though "editor support is a central pillar of language workbenches" [3], and many language workbenches do indeed support many common editor services, there is little literature on reusable formalisms and algorithms for their definition [12]. An important exception is the extensive work on defining correct refactorings (e.g., [14, 19, 16]). However, many editor services common to modern IDEs, such as reference resolution, finding declaration usages, or semantic code completion, have received little attention.

In this paper we argue that a range of semantic editor services, beyond those that have already appeared in the literature, can be specified as constraint problems. Constraints separate the declarative specification of a problem from the operational interpretations necessary to solve it. This separates concerns, but also allows reuse of constraint-based specifications for different purposes. For example, in addition to verifying the correctness of the static semantics of a program, constraint-based typing rules have also been successfully used in the implementation of semantically correct refactorings [16].

Many editor services rely on name binding information, where complex scoping and name binding rules can be a challenge for the correct implementation of editor services (e.g., correct Java refactorings involving names [15]). Although constraint-based formulations of typing rules are pervasive, constraint-based formulations of the scoping and name binding rules are rare. Name binding introduces complexities, such as avoiding accidental name capture when refactorings introduce new names. We believe that treating name binding and name resolution as an integral part of the constraint problem increases the applicability of a constraint-based approach to editor services, and can improve existing specifications from the literature in this regard.

As the basis for our investigations we use Statix, a constraint language developed for the specification of type systems [21]. Statix is built around scope graphs, a language-independent model for name binding and name resolution [20]. We argue that Statix is a suitable basis for the definition of editor services by expressing them in terms of Statix constraints and Statix type system specifications. Although Statix constraints are suitable for a declarative specification of editor services, the current deterministic solver algorithm of Statix, suitable for type checking and code navigation, is not capable of solving the editor scenarios we discuss. We identify requirements for an alternative solver for Statix that does support the interpretation and solving algorithms required for our proposed editor service definitions.

Specifically, we have the following contributions:

- We express several common editor services in terms of Statix constraint problems.
- We identify requirements on an operational semantics of Statix that is able to solve these problems.

This paper is organized as follows. Section 2 discusses the characteristics of semantic editor services and motivates our choice of editor services. In Section 3, we introduce Statix, and Statix type specifications using an example. In Section 4 we express several informing editor services in terms of the resulting program model. In Section 5 and Section 6, we do the same for the semantic code completion and extract definition refactoring editor services, respectively. In Section 7 we discuss related work. We conclude and discuss future work necessary to fully realize our proposed approach in Section 8.

2 Characterizing Editor Services

Editor services can be characterized as syntactic, those that only need the syntactic model of the program to work, and semantic, those that require the semantic model of the program [3]. We can further distinguish the semantic editor services by whether they transform the program, or merely inform the user. The informing services include editor services such as goto declaration, finding and highlighting usages, navigating to the supertype, and listing all overriding methods. The transforming editor services include quick fixes, static semantics-preserving refactorings, and (semantic) code completion.

In this section we discuss aspects that distinguish the various semantic editor services, and motivate our choice for the editor services we discuss.

Completeness

Some editor services have to be able to work on syntactically and/or semantically incomplete programs. For example, as code completion can be invoked while the user is typing, it must be able to deal with a program that is both syntactically and semantically incomplete. Similarly, the *fix import* quick fix that adds an import statement to a program to make a reference resolve, must be able to deal with programs that have incomplete semantic information; namely the program with the reference that initially does not resolve. Other editor services could provide a better user experience if they can deal with syntactically or semantically incomplete programs, but this is not a requirement.

Preserving Static Semantics

The transforming editor services all need to preserve the existing semantics of the program up to some degree. Refactorings such as *rename refactoring* and *extract definition* tend to have very strict semantic preservation requirements, including that all existing references need to resolve to the same declarations before and after the refactoring. Quick fixes and code completion, by their nature, introduce new syntax that may change certain local semantics of the code, but should not have an impact outside the area of influence.

Concrete Name Generation

Often, transforming editor services add new declarations to the program as part of their refactoring or fixing behavior. These declarations need a concrete name, one which is syntactically valid and does not clash with existing names in the program. That is, the new name should not overlap with existing names, or cause inadvertent variable capture.

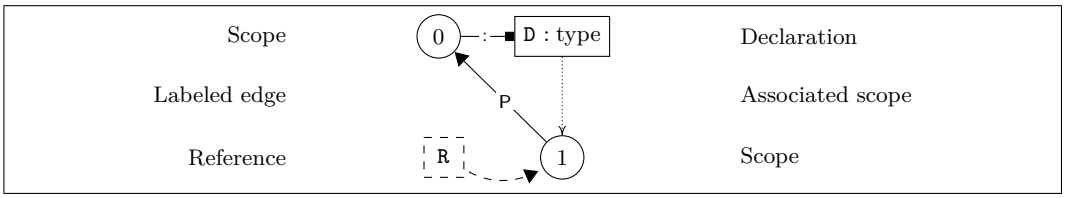


Figure 1 Overview of the notation used for scope graphs in this paper.

What We Study

Given the characterization above, we picked five editor services for which we describe the ideas of this paper. As informing editor services we choose *reference resolution*, *find usages*, and *list subclasses*, because they show how scope graphs can be used to answer these queries, where the last one requires language-specific knowledge. The last two also explore how flexible the solver must be to be able to answer such inverse queries.

We discuss two transforming editor services: *code completion*, which will have to deal with syntactically incomplete programs, and the *extract definition* refactoring, which is interesting because it introduces new syntax for which we want to use the solver to find the concrete, semantically correct, values to fill in.

We do not claim that these editor services cover all issues, or that the resulting requirements cover all editor services. However, we think that they exhibit a sufficient range of features to show the range of possibilities, and expose important requirements that need to be fulfilled to realize our approach.

3 Introduction to Statix

Statix is a recently introduced meta-language for the specification of static semantics [21], based on scope graphs and constraints [10, 20]. We chose Statix because it allows us to declare semantic editor services in terms of constraints and type system specifications.

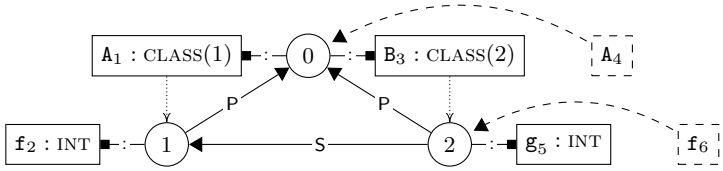
First, we explain scope graphs, a language-independent model for name binding and name resolution. Then, we introduce the rules for static semantics, and their (declarative) meaning. Finally, we explain how type checking based on these rules is implemented. We use the Java program in Figure 2 as a running example. The subscripts on program identifiers are a notational convention we use to distinguish different occurrences of the same name.

Name Binding with Scope Graphs

In Statix the name binding and resolution is part of the constraint problem, to allow complex interactions between type checking and name resolution. The name binding structure of a program is represented as a language-independent model called a *scope graph* [10, 20, 21], which is a graph of scopes and declarations in those scopes. As shown in Figure 1, the scopes are connected by labeled, directed edges. Name resolution corresponds to a query finding a path in the graph to a matching declaration.

Consider our example program and the corresponding scope graph in Figure 2. The global scope of the whole program is represented by the circled node 0. The definition of class `A` corresponds to a declaration $\texttt{A}_1$. Declarations contain both the name and its type, and therefore use the '*is of type*'-symbol ":" to label these edges. Class types are represented by the class scope. For example, scope 1 is the scope of class `A`, and its type is CLASS(1). The class scopes are lexical sub-scopes of the global scope, which is modeled by the P-labeled

```
class A1 {
    int f2 = -1;
}
class B3 extends A4 {
    int g5 = f6;
}
```



$$\text{(J-ClassDec)}\ \frac{\nabla s_c \quad s_c \xrightarrow{\mathsf{P}} s \quad s \xrightarrow{:} C_i : \text{CLASS}(s_c) \quad s_c \vdash \overline{d}\ \text{OK} \quad \mathsf{query}\ s \xmapsto{\mathsf{P}^*:} \text{DECL}(D_j)\ \mathsf{as}\ D_k : \text{CLASS}(s_d) \quad s_c \xrightarrow{\mathsf{S}} s_d}{s \vdash \texttt{class}\ C_i\ \texttt{extends}\ D_j\ \{\ \overline{d}\ \}\ \text{OK}}$$

$$\text{(J-FieldDec)}\ \frac{s \vdash [\![t]\!] \Rightarrow T \quad s \xrightarrow{:} f_j : T \quad s \vdash e : T' \quad \vdash T' <: T}{s \vdash t\ f_j = e;\ \text{OK}} \qquad \text{(J-Var)}\ \frac{\mathsf{query}\ s \xmapsto{\mathsf{P}^*\mathsf{S}^*:} \text{DECL}(x_i)\ \mathsf{as}\ x_j : T}{s \vdash x_i : T}$$

$$\text{(T-Int)}\ \frac{}{s \vdash [\![\texttt{int}]\!] \Rightarrow \text{INT}} \qquad \text{(T-Class)}\ \frac{\mathsf{query}\ s \xmapsto{\mathsf{P}^*:} \text{DECL}(C_i)\ \mathsf{as}\ C_j : T}{s \vdash [\![C_i]\!] \Rightarrow T}$$

$$\text{(<:-Int)}\ \frac{}{\vdash \text{INT} <: \text{INT}} \qquad \text{(<:-Class)}\ \frac{\mathsf{query}\ s_1 \xmapsto{\mathsf{S}^*} \text{SCOPE}(s_2)\ \mathsf{as}\ p}{\vdash \text{CLASS}(s_1) <: \text{CLASS}(s_2)}$$

Figure 2 Example Java program with two classes, its corresponding scope graph, and the relevant Statix typing rules.

(parent) edges. The fact that class `B` extends class `A` is represented by the edge labeled S (supertype). This edge makes the fields from the super class visible in the subclass, but is also used to decide subtyping between class types. The field declarations are similar to the class declarations, but in the class scopes.

Resolving a name corresponds to querying the scope graph for a matching declaration. Resolution queries are parameterized by a regular expression that determines which declaration can be reached, a predicate determining which declarations match. An additional order on labels is used to disambiguate multiple matching declarations. For example, the class reference $\texttt{A}_4$ is resolved in the global scope 0. Class references are resolved in the lexical context, and the regular expression that encodes this is $\mathsf{P}^*:$, which matches any path to a declaration via any number of P-steps to lexical parents. The declaration itself should match the reference, which is specified with the predicate $\text{DECL}(\texttt{A}_4)$, which holds for any x_i where $x = \texttt{A}$. In this case the reference resolves directly to declaration $\texttt{A}_1$ in scope 0.

Resolving the variable reference $\texttt{f}_6$ follows the same pattern. However, it should be possible to resolve not just to variables in the lexical context, but also to fields in the super class. This is achieved by using the regular expression $\mathsf{P}^*\mathsf{S}^*:$. This allows the reference to be resolved to declaration $\texttt{f}_2$, by following the S-edge to scope 1.

Type Specifications

The rules of a Statix specification formally describe the scope graph that corresponds to a program, as well as constraints on references and types, in terms of syntax-directed rules. Figure 2 shows some of the rules that apply to our example program. For example, the rule (J-ClassDec) specifies that a class definition c is well-formed in scope s, written as $s \vdash c$ OK, if the scope graph has the correct structure, and the definitions in the class are well-formed as well ($s_c \vdash \overline{d}$ OK). The first three premises state that the scope graph contains a scope s_c that is unique to this class (∇s_c), that this scope has a P-edge to its lexical parent ($s_c \xrightarrow{\mathsf{P}} s$), and that there is a declaration C_i for the class in the lexical scope s, typed by the class scope s_c ($s \xrightarrow{\vdots} C_i : \text{CLASS}(s_c)$). The last two premises say that the reference to the super class resolves to a declaration D_k, which is typed by a class scope s_d ($\mathsf{query}\ s \xmapsto{\mathsf{P}^*\vdots} \text{DECL}(D_j)\ \mathsf{as}\ D_k : \text{CLASS}(s_d)$), and that an inheritance edge exists from the scope of this class to the scope of the super class ($s_c \xrightarrow{\mathsf{S}} s_d$).

The rule (J-FieldDec) specifies that a field declaration is well-formed if a declaration for the field exists in the scope graph ($s \xrightarrow{\vdots} f_j : T$), if the assigned expression is well-typed for some type T' ($s \vdash e : T'$), and the expression type T' is a subtype ($\vdash T' <: T$) of the semantic type T corresponding to the type annotation ($s \vdash [\![t]\!] \Rightarrow T$). The relations for semantic typing, subtyping, and expression typing are also defined with Statix rules. The only built-in constraints are constraints to define the scope graph, constrains to query the scope graph, and term equality. All other relations are completely determined by the rules from the specification.

Type Checking

The specification is declarative, and only gives a logical description of what well-formed programs are with respect to a scope graph. We made no assumptions yet on how to operationalize it. One possible interpretation is to use the specification to type check programs. Checking that a program p is well-formed corresponds to checking if the constraint $s \vdash p$ OK is satisfiable. Van Antwerpen et al. describe an algorithm to solve such constraints, given a specification and a program p as input [21]. The algorithm uses the rules from the specification to simplify constraints until only built-in constraints remain. These are solved using unification and scope graph resolution algorithms. This solver is deterministic: it does not use back-tracking, and only applies rules if they match the given program construct. The result of solving a constraint such as $s \vdash p$ OK is a solution consisting of a variable assignment V and a scope graph G, or no solution if the constraint cannot be satisfied. A resulting program model would also include the types assigned to all expressions, and the resolution R of all references in the program.

4 Informing Editor Services

Many editors have editor services through which the user can navigate their program. The simplest of these involve clicking a reference and jumping to the corresponding declaration, or listing all usages of a declaration, but there are also more sophisticated editor services such as those that list the subclasses of a particular class. All these services have in common that they can be expressed as queries on the program model that resulted from type checking. Even though these queries themselves do not change the program, they may be part of the implementation of other editor services that do change the program. For example, a refactoring that renames a variable first needs to find all usages of the variable to ensure they are all renamed.

Reference resolution and finding declaration usages can easily be derived from the program model, which contains the resolution relation R, which consists of pairs of references and their declaration. Consider the example in Figure 2 again. Finding the declaration corresponding to reference $\texttt{A}_4$, involves finding the entry for the reference in R. Conversely, finding all usages of declaration $\texttt{f}_2$ corresponds to a reverse lookup. These queries parallel the resolution queries in the typing rules, and can directly be derived from the specification.

While a query to find all subclasses of a certain class is not directly present in the typing rules, we can phrase such a query as a constraint, which we solve with respect to the given program model. For instance, how would we specify – in constraints – the query to get all subclasses of class $\texttt{A}_1$? We assume as input the declaration itself, and the scope 0 of the class definition, which should be part of the program model. The general idea of the query is to find the class scope, find other class scopes that are connected to it by inheritance edges, and find their corresponding declarations. This is encoded by the following constraint:

$$\textsf{query}\ 0 \mapsto \text{DECL}(\texttt{A}_1)\ \textsf{as}\ \texttt{A}_1 : \text{CLASS}(s_c)$$

$$\textsf{query}\ s_d \xmapsto{\text{S}^+} \text{SCOPE}(s_c)$$

$$\textsf{query}\ s' \mapsto \text{TRUE}\ \textsf{as}\ x_i : \text{CLASS}(s_d)$$

where s_c, s_d and s' are existentially quantified, and x_i is the output. The first constraint says that the class declaration is typed by a scope s_c. The second constraint states that there is a path from some subclass scope s_d to the class scope s_c. The final constraint indicates that there is a declaration with any name x_i, which is typed by the subclass scope s_d.

None of these constraints appear as such in the typing rules, and we have to do work to find possible solutions. This may seem daunting, given the free variables for scopes and names, both of which have infinite domains. However, we are only interested in solutions that are valid in the context of an existing scope graph. This scope graph is always finite, which gives us an initial, if maybe inefficient, strategy to find possible solutions. In the case of our example, there is one solution, where $s_c = 1$, $s_d = 2$, $s' = 0$, and $x_i = \texttt{B}_3$.

The given formulation requires an algorithm quite different from the current, deterministic solver of Statix. Instead of strictly relying on inference via forward resolution and unification, it needs to be able to guess values, try different alternatives, and back-track on failed attempts. An alternative approach could have been to change scope graph queries to allow backward edge steps. For example, if we use $\hat{l}$ for backward steps in the regular expression, our second constraint might have been:

$$\textsf{query}\ s_c \xmapsto{\text{S}^+} \text{TRUE}\ \textsf{as}\ s_d$$

In this case, we could do forward resolution from scope s_c again, reusing the resolution algorithm that is already there. Although this approach may work for queries designed specifically with editor services in mind, it does not work if we want to use our typing rules as-is. Therefore, we choose not to change the formalism, but require a solver that supports more flexible inference.

Summary

We showed that queries on the program model can be expressed as constraints, and that finding answers to these queries corresponds to solving these constraints in the context of a given program model. We discussed that solving these queries requires different solver strategies to be supported by the solver for Statix. However, this solver would be independent

of the specific object language the query is for, and is therefore reusable between languages. Given such a solver, implementing such editor services reduces to being able to specify the query as a constraint.

5 Code Completion

Code completion is an editor service that suggests a valid code fragment to be inserted at the caret position. This assists the user while typing, attempts to minimize typing errors, and aids in discovery by showing the possible syntax and references. Syntactic code completion is the most basic kind of code completion: it suggests only syntax fragments that fit at the caret location, with no regard for whether the proposal fits semantically. Semantic code completion improves on this by suggesting only those proposals that conform to the static semantics of the language, such as only suggesting expression syntax that can produce a value of the expected type. Additionally, semantic code completion proposes inserting references to declarations, such as variables, fields, and functions, that are visible from the scope at the caret location. In this section we discuss how the type system and semantic specification of a language can be used to provide accurate semantic code completion without additional work on the part of the language designer.

In Figure 3 we show an example Java program with the caret position denoted by |, near the end of the last line of class $\texttt{X}_{11}$. The program is incomplete: it is not syntactically valid because the user has not yet finished typing. Despite this, we would want the semantic model of the program so we can suggest relevant syntax and references.

As a first step, we propose to use the techniques described by Amorim et al. in [2] to use the syntactic specification of the language to introduce *placeholders* into the abstract syntax. A placeholder is a term in the syntax that represents a place where syntax of a certain sort, such as an expression or a declaration, could be inserted. This makes the program syntactically complete, and the placeholders provide us with syntax terms which we can constrain. Therefore, to the completion service, the incomplete line of code has the following syntax, with placeholder ***$Exp*** for a possible expression that would complete the program:

```
int i13 = $Exp ;
```

At this point, we would want to invoke the solver and let it verify our program using the rules shown in Figure 3. However, no rules apply to the placeholder term ***$Exp***. Instead, we propose to replace any occurrence of a placeholder in the syntax terms with a corresponding constraint variable in the constraint terms. In this example, we use ε for ***$Exp***, which, because of the semantic rule (J-FieldDec), results in the following constraints for this line:

$$3 \vdash [\![\texttt{int}]\!] \Rightarrow T \qquad 3 \xrightarrow{\vdots} \texttt{i}_{13} : T \qquad 3 \vdash \varepsilon : T' \qquad \vdash T' <: T$$

Solving these constraints assigns $T' \mapsto \textsc{int}$ and $T \mapsto \textsc{int}$. In other words, the editor service has inferred that the expected type of the expression on that line must be INT, and produced the scope graph shown in Figure 3. The solver can continue, trying to find an assignment for ε. There are two rules in Figure 3 that it could apply: (J-Plus) and (J-ThisMethodCall). In fact, we would want the solver to return both solutions for code completion. We will explore both these alternatives.

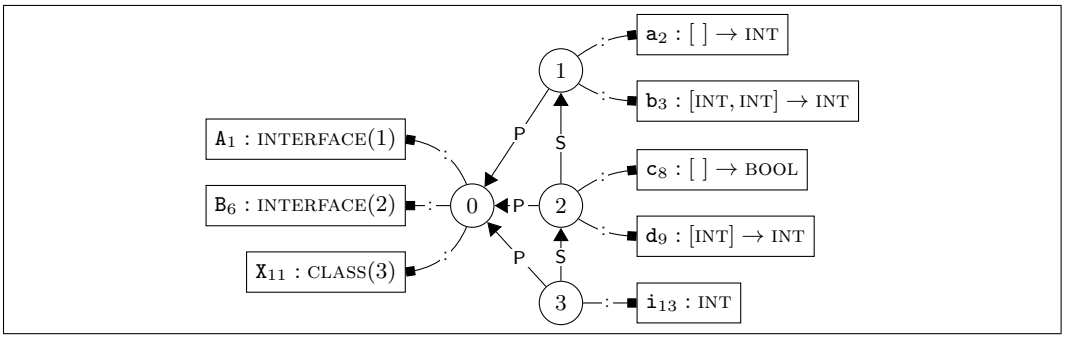
Expression Completion

From rule (J-Plus) $(s \vdash e_1 + e_2 : T)$ we would get the assignment $\varepsilon = \varepsilon_1 + \varepsilon_2$, where ε_1 and ε_2 are new constraint variables introduced by the solver. We would like to stop here, and let the solver return the solution $\varepsilon = \varepsilon_1 + \varepsilon_2$. Note that this solution is incomplete: it does not

```
interface A1 {
    int a2();
    int b3(int x4, int y5);
}

interface B6 extends A7 {
    boolean c8();
    int d9(int x10);
}

class X11 implements B12 {
    int i13 = |;
}
```



$$\text{(J-InterfaceDec)}\ \frac{\nabla s_c \quad s_c \xrightarrow{\mathsf{P}} s \quad s \xrightarrow{:} C_i : \text{INTERFACE}(s_c) \quad \text{query } s \mapsto \text{DECL}(\overline{D_j}) \text{ as } \{\overline{D_k} : \text{INTERFACE}(\overline{s_d})\} \quad s_c \xrightarrow{\mathsf{S}} \overline{s_d} \quad s_c \vdash \overline{d}\ \text{OK}}{s \vdash \texttt{interface}\, C_i\, \texttt{extends}\, \overline{D_j}\ \{\ \overline{d}\ \}\ \text{OK}}$$

$$\text{(J-InterfaceMethodDec)}\ \frac{s \vdash [\![C_i]\!] \Rightarrow T \quad s \vdash [\![\overline{C_k}]\!] \Rightarrow \overline{T_k} \quad s \xrightarrow{:} x_j : \overline{T_k} \to T}{s \vdash C_i\ x_j(\overline{C_k}\ \overline{x_k}); \text{OK}}$$

$$\text{(J-ClassDec)}\ \frac{\nabla s_c \quad s_c \xrightarrow{\mathsf{P}} s \quad s \xrightarrow{:} C_i : \text{CLASS}(s_c) \quad s_c \vdash \overline{d}\ \text{OK} \quad \text{query } s \mapsto \text{DECL}(\overline{D_j}) \text{ as } \{\overline{D_k} : \text{INTERFACE}(\overline{s_d})\} \quad s_c \xrightarrow{\mathsf{S}} \overline{s_d}}{s \vdash \texttt{class}\, C_i\, \texttt{implements}\, \overline{D_j}\ \{\ \overline{d}\ \}\ \text{OK}}$$

$$\text{(J-FieldDec)}\ \frac{s \vdash [\![t]\!] \Rightarrow T \quad s \xrightarrow{:} f_j : T \quad s \vdash e : T' \quad \vdash T' <: T}{s \vdash t\ f_j = e;\ \text{OK}}$$

$$\text{(J-ThisMethodCall)}\ \frac{s \vdash \overline{e} : \overline{V} \quad \vdash \overline{V} <: \overline{U} \quad \text{query } s \xrightarrow{\mathsf{S}^*:} \text{DECL}(m_i) \text{ as } \{m_j : \overline{U} \to T\}}{s \vdash m_i(\overline{e}) : T}$$

$$\text{(J-Plus)}\ \frac{s \vdash e_1 : T_1 \quad s \vdash e_2 : T_2 \quad T_1 = T_2 = T = \text{INT}}{s \vdash e_1 + e_2 : T}$$

Figure 3 Java program illustrating code completion, and the corresponding scope graph and relevant Statix typing rules.

describe the whole program as there are still free constraint variables in them. Therefore, the solver would need to be able to return incomplete solutions. As part of this solution, we get some constraints that not ground because they contain these free constraint variables:

$$3 \vdash \varepsilon_1 : \text{INT} \qquad\qquad 3 \vdash \varepsilon_2 : \text{INT}$$

Translated back to syntax terms, replacing the free constraint variables by placeholders, this would result in the following syntax on the line being completed:

```
int i13 = $Exp + $Exp ;
```

Of course, we could also let the solver continue its search to find assignments for ε_1 and ε_2, but this would likely result in an ever expanding sequence of $\varepsilon_1 + \varepsilon_2 + \varepsilon_3 + \ldots$. Ultimately, there are infinitely many solutions if we were to try to make all variables ground. This shows that we need a way to instruct the solver on how deep we want a constraint variable to be solved. In this example, we want solutions for ε only one level deep.

Method Call Completion

When the solver instead applies rule (J-ThisMethodCall), we get method call completion: where code completion suggests calls to methods in scope at the caret position, and whose return a type is compatible with the expected type of the expression. From rule (J-ThisMethodCall) $(s \vdash m_i(\overline{e}) : T)$ we would get the assignment $\varepsilon = \mu(\overline{\varepsilon})$, again introducing new constraint variables μ and $\overline{\varepsilon}$ to represent the method name and arguments respectively.

Since proposing just the syntax for a method call is not very satisfactory to a user, this time we *do* want to get another level of solutions. At least, we want μ to be solved, but we do not care about $\overline{\varepsilon}$. We need a way to indicate this to the solver. Through the rule (J-ThisMethodCall) the solver would add these constraints:

$$3 \vdash \overline{\varepsilon} : \overline{V} \qquad\qquad \vdash \overline{V} <: \overline{U} \qquad\qquad \mathsf{query}\ 3 \xmapsto{\mathsf{S}^*:} \text{DECL}(\mu)\ \mathsf{as}\ \{m_j : \overline{U} \to \text{INT}\}$$

There are multiple possible assignments for constraint variables μ and $\overline{\varepsilon}$, and for code completion to work, the solver must find them all. The following table shows the possible assignments for μ, $\overline{\varepsilon}$, T, $\overline{U}$, and $\overline{V}$ that the solver might yield.

Solution	μ	$\overline{\varepsilon}$	T	$\overline{U}$	$\overline{V}$
Solution 1	$\mathtt{a}_2$	[ ]	INT	[ ]	[ ]
Solution 2	$\mathtt{b}_3$	$[\varepsilon_1, \varepsilon_2]$	INT	[INT, INT]	$[\tau_1, \tau_2]$
~~Solution 3~~	$\mathtt{c}_8$	[ ]	BOOL	[ ]	[ ]
Solution 4	$\mathtt{d}_9$	$[\varepsilon_1]$	INT	[INT]	$[\tau_1]$

Note that solution 3 is not valid, as it tries to assign $T \mapsto$ BOOL whereas T had previously already been assigned INT. Also note how the solver could infer lists of constraint variables for $\overline{U}$ and $\overline{V}$. But, as before, we would not want the solver to keep expanding on the constraint variables it has introduced. If we had not relaxed these variables such that they may remain free, the solver would have to find some assignment for the variables that satisfies them. In this example the solver might have added a method call to an arbitrary method with a compatible return type, such as $\mathtt{a}_2$. In other scenarios the solver may not be able to find such a solution, or find infinitely many.

The solutions returned by the solver can be turned into syntax fragments and presented to the user as code completion proposals, where we replace the free constraint variables by syntax placeholders. The order of the proposals is not determined by the solver, as we

consider this to be a separate concern. For example, we may want to order the proposals by their frequency of use, or use the semantic model to order the proposals by closeness (e.g., local variables before global variables). In this example, code completion would propose the following method calls:

```
a2()
b3($Exp, $Exp)
d9($Exp)
```

Summary

To use the semantic of the programming language for code completion, we first need a semantic specification that includes a model for name binding. This is already provided by the scope graphs used by the Statix constraint solver. However, the solver also needs to support returning incomplete solutions. The solver needs to be able to distinguish between constraint variables that we want to have solved and those that may remain free, and we need to be able to indicate how deep we want a given constraint variable to be solved. By using the semantic rules, a solution can include syntactic assignments to variables. Finally, the solver must be able to return more than one solution, so we can display them all to the user as part of code completion.

6 Extract Definition

A common refactoring is the *extract definition* refactoring, where the user selects a subexpression and the refactoring replaces any occurrences of that expression by a reference to a variable definition initialized by the subexpression. In the example in Figure 4, we want to extract the `x - 3` subexpression into a separate definition. We assume the program is syntactically complete and semantically correct.

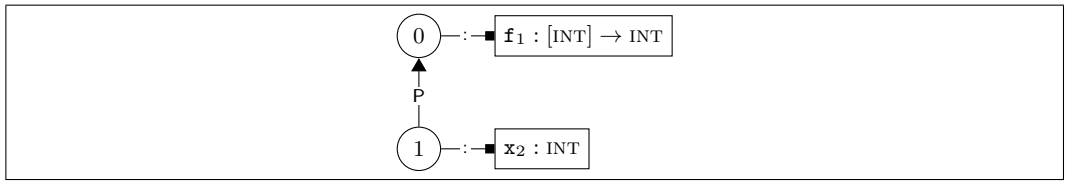
The first step in this refactoring is to determine the new syntax that we expect as a result of the refactoring. This is language-specific syntax, selected by the user and specified in advance by the language developer. The syntax fragment uses placeholders, as shown below, where *$Type* is a placeholder for the type of the newly created variable and *$ID* is a placeholder for a variable name. In this case we want all three occurrences *$ID* to refer to the same variable.

```
int f1(int x2) {
    $Type $ID = x3 - 3;
    return $ID + $ID * x5;
}
```

We create a copy of the previous, valid, solution returned by the solver, and adapt it to this refactoring. This is a two-part process: relaxing the solution, and adding new constraints to the problem. Relaxing the solution removes any variables, resolutions, constraints, and scope graph nodes that are no longer valid or that impact the aspects we want to refactor. For extracting a definition, relaxation only involves removing the reference relation $x_4 \mapsto x_2$, since the reference x_4 has been removed. However, we still want the variable references x_3 and x_5 to resolve to the same definition x_2.

Now we can add new constraints to the problem, but the refactoring should add only those constraints that result from the changed syntax. The constraints may contain syntax terms, but we replace any occurrences of the placeholders by constraint variables. For the

```
int f1(int x2) {
    return (x3 - 3) + (x4 - 3) * x5;
}
```



$$\text{(J-MethodDec)}\ \frac{\begin{array}{c} s \vdash [\![C_i]\!] \Rightarrow T \quad \nabla s_m \quad s_m \xrightarrow{\mathsf{P}} s \\ s \vdash [\![\overline{C_k}]\!] \Rightarrow \overline{T_k} \quad s_m \xrightarrow{:} \overline{p_k} : \overline{T_k} \\ s_m \vdash \bar{b}\ \text{OK} \quad s \xrightarrow{:} m_j : \overline{T_k} \to T \\ s_m \vdash e : U \quad \vdash U <: T \end{array}}{s \vdash C_i\ m_j(\overline{C_k}\ \overline{p_k})\ \{\ \bar{b}\ \texttt{return}\, e;\}\ \text{OK}}$$

$$\text{(J-VarDec)}\ \frac{s \vdash [\![C_x]\!] \Rightarrow T \quad s \xrightarrow{:} v_y : T \quad s \vdash e : T}{s \vdash C_x\ v_y = e;\ \text{OK}}$$

$$\text{(J-Var)}\ \frac{\mathsf{query}\ s \xmapsto{\mathsf{P}^*\mathsf{S}^*:} \text{DECL}(x_i)\ \mathsf{as}\ x_j : T}{s \vdash x_i : T}$$

$$\text{(J-IntBinOp)}\ \frac{s \vdash e_1 : T_1 \quad s \vdash e_2 : T_2 \quad T_1 = T_2 = T = \text{INT}}{s \vdash e_1 \oplus e_2 : T}$$

Figure 4 Java program before applying the *extract definition* refactoring, and the corresponding scope graph and relevant Statix typing rules.

type placeholder *$Type*, we will use the constraint variable τ. Since we want all occurrences of the *$ID* placeholder to refer to the same variable, we should replace all occurrences with the same constraint variable.

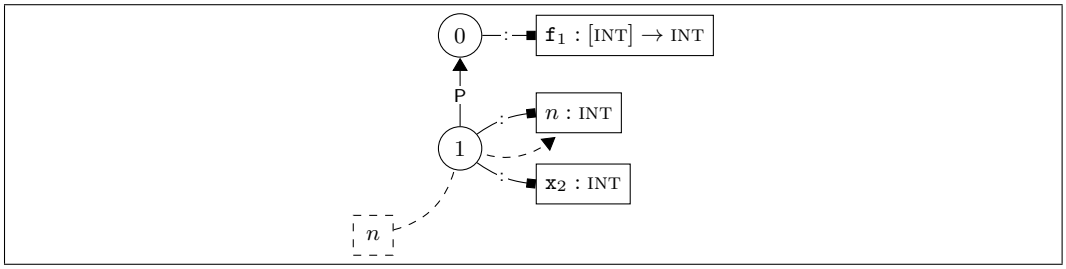
Where other approaches use the solver to find a concrete name for the variable ([16]), we argue that this is not necessary for the solver to give a correct result. We merely want to indicate to the solver that the new name is different from all other names in the program. This gives a separation of concerns: the solver can verify that the program satisfies the constraints without needing to produce any concrete names, and generating the concrete names can be done externally after the solver has verified the program. For example, some IDEs provide a list of name suggestions that they generate from the context, such as the type of the expression.

To distinguish the abstract name from any other name in the program, we can use a rigid variable: one that is distinct from any other variable or name. Similar to how rigid variables are used to create new distinct scopes in the scope graph (through ∇s), we create a new rigid variable to represent the name of the newly introduced variable: ∇n. Due to rules (J-VarDec) and (J-Var), this results in the following constraints:

$$\begin{array}{ll} \nabla n & \\ 1 \vdash [\![\tau]\!] \Rightarrow T & 1 \xrightarrow{:} n : T \\ 1 \vdash (\mathtt{x}_3 - 3) : T & \mathsf{query}\ 1 \xmapsto{\mathsf{P}^*\mathsf{S}^*:} \text{DECL}(n)\ \mathsf{as}\ x_j : T \end{array}$$

Solving these constraints results in the variable assignment and scope graph shown in Figure 5.

```
int f(int x) {
    int n = x - 3;
    return n + n * x;
}
```



	n	τ	T	v_j
Solution	n	`int`	INT	n

Figure 5 Java program after applying the *extract definition* refactoring, and the corresponding scope graph and variable assignments. Note that n in the program is a rigid variable, which has yet to be assigned a concrete name.

From this we can conclude that the refactoring is valid, does not semantically change the program (since the existing constraints and reference resolutions are preserved), and that the type of the newly introduced variable is `int`. However, to finish the refactoring we have to decide on a concrete name for rigid variable n. A concrete name can be provided by the refactoring tool or by the user. In any case, we can test whether the suggested name is allowed by reinvoking the solver with the new solution and one additional constraint: to constrain n to the chosen name, say `i`.

$$n = \texttt{i}$$

The new constraint may result in an invalid solution, for example when the chosen name overlaps with another, or causes inadvertent name capture somewhere in the program. In this example, `x` is not allowed as a concrete name for n. However, if this results in a valid solution, the concrete name is acceptable and the refactoring can finish. In this example it would produce the following code:

```
int f(int x) {
    int i = x - 3;
    return i + i * x;
}
```

Summary

As part of the refactoring we generate new syntax, where we use placeholders to indicate where we need more information. In the newly generated constraints we have a constraint variable taking the place of every placeholder, which allows us to use the solver to find a solution to the problem. By using a rigid variable in place of a concrete name, we can indicate that the name is different from all other names in the program without having to specify such a name concretely, giving a separation of concerns between finding whether the program is valid and what concrete name to choose.

7 Related Work

Erdweg et al. [3] identify editor services as an important aspect of language workbenches, and give an overview of commonly supported editor services. However, Omar et al. have pointed out that the study of the semantic foundations of editors and editor interactions has received little attention so far [12]. We discuss work related to code completion, and refactoring, as those are most relevant to the editor services we covered in this paper.

Reference Resolution

Language workbenches such as Xtext [4] and Spoofax [6, 22] provide support for language-parametric reference resolution based on declarative name binding specifications. Xtext supports specification of references in the language grammar as crosslinks, which specify the sort that an identifier can refer to. Xtext will check the validity of the references and add them to the model.

The first approach to declarative name binding specification in Spoofax was the NaBL name binding language [7]. The name binding rules defined the definition sites and their scopes based on the abstract syntax of the program. The built-in reference resolution algorithm could only create an index in which references can be looked up, which limits its flexibility to be used in other editor services.

Instead, the approach we use in this paper uses the expressiveness of the constraints and the flexibility of the Statix constraint solver to enable reference resolution to be used in various editor services.

Code Completion

The Xtext and Spoofax language workbenches also provide support for language-parametric syntactic completion, based on a syntax definition. In the case of Xtext, it suggests possible keywords. Spoofax suggests complete syntactic constructs, and represents incomplete syntax trees using placeholders that act like holes in the program text [2]. This representation is instrumental for translating an incomplete program to an abstract syntax tree with variables, which allows us to use it in a constraint context. A program with placeholders is similar to the representation of an AST with holes that is common in structure editors. Although structure editors are primarily concerned with guaranteeing that the program is well-typed with respect to the abstract syntax signature, recent work investigates editors that also maintain other well-formedness properties, such as well-typedness.

The Hazelnut editor provides a language-parametric structure editor that guarantees well-typed ASTs for languages whose type system is defined in a bidirectional style [11]. JastAdd extends their reference attribute grammars to provide a context-sensitive completion service that suggest the names of variables and functions, but still requires some language-specific effort to derive these suggestions [18]. Steimann et al. use constraint-based language specifications to ensure edits preserve well-formedness [17]. They focus on an architecture that allows interaction between the solver and the user during the editing process, to resolve conflicts that may have been introduced. Our aim is to generate semantic completion proposals by combining the mechanisms for syntactic completion, with checking and inference based on the language specification. Another important difference is that issues around name resolution are largely ignored in their work, because references are actual references in the underlying model, whereas in our text-based setting we need to consider naming issues.

Semantic code completion also has similarities to interactive proof search, such as offered by proof assistants. For example, the editor of Agda [1] features holes that are similar to placeholders. An automatic procedure tries to find proof terms (expressions) that fit the goal (type). There are some important differences with our approach to code completion. The procedure to find these terms is not language-parametric, but specific to Agda. The search procedure does not exploit the typing rules, but duplicates knowledge from the type checker. Type correctness is guaranteed by type checking the fragment after it is generated.

Refactoring

There is a long line of research on the specification and implementation of refactorings. Tip et al. [19] study type related refactorings, such as adding type parameters, extracting interfaces, and pulling up methods. They use type constraints to specify the invariants that ensure correct behavior. Steimann and others [16] extend this work to include constraints for other aspects such as access modifiers and names. By representing the program itself using constraint variables, both the invariants and the refactoring intent can be represented as constraints. Finding the refactored program, within the limits of the given constraints, is delegated to the constraint solver. This approach is in many aspects similar to ours, and hopefully techniques they developed for performance carry over to our approach. An important difference is that by using Statix, the constraint solver is aware of the complete binding model. In their approach preventing capture requires the introduction of inequality constraints between names. These constraints do not follow from regular constraint-based typing rules. In our approach, the resolution constraints that are part of the typing rules can also be used to ensure the invariance of name resolution during refactoring.

8 Conclusion

In this paper we have discussed various semantic editor services, and shown how they can be expressed in terms of the semantic rules, constraints, and scope graph. We show that Statix constraints are expressive enough to formulate interesting editor services. We have pointed out that the Statix solver used for type checking is not suitable for the scenarios that arise in editor services, and we have identified several requirements that such alternative solver strategies should have. The main requirements we identified are:

- The solver must be able to try different alternatives, guess values, back-track on failed attempts, and able to return multiple solutions. As we have shown, this is a requirement to implementing code completion, but also for other editor services such as find usages and find all subclasses.
- Instead of always searching for complete solutions (i.e., assignments to all variables), the search should be controlled by user-defined criteria, including whether the solver should only consider deterministic inference, or whether it tries to find solutions non-deterministically. These criteria should be able to depend on variables appearing in the constraints. Specifying which constraint variables may remain unconstrained, and how deep the solver should search for an assignment, allows us to direct the search to finding solutions to only those constraint variables we are interested in, and prevent the solver from getting stuck.

Finally, we propose to extend the mechanisms of creating scopes in Statix to a general mechanism of rigid variables. These rigid variables can be used to solve problems around inventing new concrete names in the constraint solver, and should make it easier to implement various refactorings without having to deal with concrete names, accidental variable

capture, and ambiguous names. Factoring out the choice of finding concrete names separates concerns, and also allows for language-specific strategies (e.g., suggesting variables names based on types).

Future Work

This paper presented ideas on what would be needed for language-parametric semantic editor services. To verify our approach, we need to implement the proposed extensions to the Statix solver. This will allows us to evaluate their feasibility in practice, as it is not clear whether implementing some of these techniques, such as having the solver back-track and trying to find multiple possible solutions, would cause performance issues or introduce non-termination. And if so, how we could avoid that without impacting the expressiveness of the semantic rules too much.

There are editor services, other than those we discussed, to which we could apply our approach, such as *fix import*, *search for symbol*, and in particular *rename refactoring*. Rename refactoring is interesting because it not only needs to rename the references to the renamed declaration, but possibly other references and declarations as well. For example, when a method is renamed, all overriding methods need to be renamed too, and this relation is not visible in the program model, but only encoded in the semantic rules.

While our current approach is focussed on preserving the static semantics of the program, for certain refactorings it may be required to extend the approach to also preserve certain dynamic aspects of the semantics. Additionally, a combination of our approaches might be used to implement program generation that is guaranteed to produce programs that are semantically correct.

References

1 Catarina Coquand, Makoto Takeyama, and Dan Synek. An Emacs-Interface for Type-Directed Supportfor Constructing Proofs and Programs. In *European Joint Conferences on Theory and Practice of Software, ENTCS*, volume 2, 2006.

2 Luis Eduardo de Souza Amorim, Sebastian Erdweg, Guido Wachsmuth, and Eelco Visser. Principled syntactic code completion using placeholders. In Tijs van der Storm, Emilie Balland, and Dániel Varró, editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands, October 31 - November 1, 2016*, pages 163–175. ACM, 2016.

3 Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. The State of the Art in Language Workbenches - Conclusions from the Language Workbench Challenge. In Martin Erwig, Richard F. Paige, and Eric Van Wyk, editors, *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings*, volume 8225 of *Lecture Notes in Computer Science*, pages 197–217. Springer, 2013. doi:10.1007/978-3-319-02654-1_11.

4 M. Eysholdt and H. Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 307–309. ACM, 2010.

5 Martin Fowler. Language Workbenches: The Killer-App for Domain Specific Languages?, 2005.

6 Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, pages 444–463, Reno/Tahoe, Nevada, 2010. ACM. doi:10.1145/1869459.1869497.

7 Gabriël Konat, Lennart C. L. Kats, Guido Wachsmuth, and Eelco Visser. Declarative Name Binding and Scope Rules. In Krzysztof Czarnecki and Görel Hedin, editors, *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers*, volume 7745 of *Lecture Notes in Computer Science*, pages 311–331. Springer, 2012. doi:10.1007/978-3-642-36089-3_18.

8 Tom Mens, Serge Demeyer, Bart Du Bois, Hans Stenten, and Pieter Van Gorp. Refactoring: Current Research and Future Trends. *Electronic Notes in Theoretical Computer Science*, 82(3):483–499, 2003.

9 Gail C. Murphy, Mik Kersten, and Leah Findlater. How Are Java Software Developers Using the Eclipse IDE? *IEEE Software*, 23(4):76–83, 2006. doi:10.1109/MS.2006.105.

10 Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A Theory of Name Resolution. In Jan Vitek, editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 205–231. Springer, 2015. doi:10.1007/978-3-662-46669-8_9.

11 Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. Hazelnut: a bidirectionally typed structure editor calculus. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 86–99. ACM, 2017.

12 Cyrus Omar, Ian Voysey, Michael Hilton, Joshua Sunshine, Claire Le Goues, Jonathan Aldrich, and Matthew A. Hammer. Toward Semantic Foundations for Program Editors. In Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi, editors, *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA*, volume 71 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPIcs.SNAPL.2017.11.

13 Martin P. Robillard, Wesley Coelho, and Gail C. Murphy. How Effective Developers Investigate Source Code: An Exploratory Study. *IEEE Trans. Software Eng.*, 30(12):889–903, 2004. doi:10.1109/TSE.2004.101.

14 Max Schäfer and Oege de Moor. Specifying and implementing refactorings. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, pages 286–301, Reno/Tahoe, Nevada, 2010. ACM. doi:10.1145/1869459.1869485.

15 Max Schäfer, Andreas Thies, Friedrich Steimann, and Frank Tip. A Comprehensive Approach to Naming and Accessibility in Refactoring Java Programs. *IEEE Trans. Software Eng.*, 38(6):1233–1257, 2012. doi:10.1109/TSE.2012.13.

16 Friedrich Steimann. Constraint-Based Refactoring. *ACM Transactions on Programming Languages and Systems*, 40(1), 2018. doi:10.1145/3156016.

17 Friedrich Steimann, Marcus Frenkel, and Markus Voelter. Robust projectional editing. In Benoît Combemale, Marjan Mernik, and Bernhard Rumpe, editors, *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017, Vancouver, BC, Canada, October 23-24, 2017*, pages 79–90. ACM, 2017. doi:10.1145/3136014.3136034.

18 Emma Söderberg and Görel Hedin. Building semantic editors using JastAdd: tool demonstration. In Claus Brabrand and Eric Van Wyk, editors, *Language Descriptions, Tools and Applications, LDTA 2011, Saarbrücken, Germany, March 26-27, 2011. Proceeding*, page 11. ACM, 2011. doi:10.1145/1988783.1988794.

19 Frank Tip, Robert M. Fuhrer, Adam Kiezun, Michael D. Ernst, Ittai Balaban, and Bjorn De Sutter. Refactoring using type constraints. *ACM Transactions on Programming Languages and Systems*, 33(3):9, 2011. doi:10.1145/1961204.1961205.

20 Hendrik van Antwerpen, Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A constraint language for static semantic analysis based on scope graphs. In Martin Erwig and Tiark Rompf, editors, *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 49–60. ACM, 2016. doi:10.1145/2847538.2847543.

21 Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. Scopes as types. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA), 2018. doi:10.1145/3276484.

22 Eelco Visser, Guido Wachsmuth, Andrew P. Tolmach, Pierre Néron, Vlad A. Vergu, Augusto Passalaqua, and Gabriël Konat. A Language Designer's Workbench: A One-Stop-Shop for Implementation and Verification of Language Designs. In Andrew P. Black, Shriram Krishnamurthi, Bernd Bruegge, and Joseph N. Ruskiewicz, editors, *Onward! 2014, Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, part of SPLASH '14, Portland, OR, USA, October 20-24, 2014*, pages 95–111. ACM, 2014. doi:10.1145/2661136.2661149.