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

New programming languages often lack good IDE support, as developing advanced semantic editor services takes additional effort. In previous work we discussed the operational requirements of a constraint solver that leverages the declarative type system specification of a language to provide language-parametric semantic editor services. In this work we describe the implementation of our solver as a two stage process: inference and search. An editor-service specific search strategy determines how and where the search is conducted, and when it terminates. We are currently implementing and evaluating this idea.

CCS Concepts • **Software and its engineering → Semantics**.

Keywords constraint programming, semantics, constraint solving, Spoofax, Statix, editor services

ACM Reference Format:
Daniel A. A. Pelsmaeker, Hendrik van Antwerpen, and Eelco Visser. 2019. Towards Language-Parametric Semantic Editor Services Based on Declarative Type System Specifications. In *Proceedings of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion '19), October 20–25, 2019, Athens, Greece.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3359061.3362782

1 Introduction

When creating a new programming language, it takes additional effort to provide good editor services for the language in an IDE, which is important for effective comprehension, navigation, and refactoring of the code. While existing work addresses the problem of supporting a language across multiple IDEs, such as using Language Server Protocol [1], Monto [3], AESI [6], and MagpieBridge [4], none of them address the difficulty of implementing the editor services.

In previous work [7] we argued that we can specify advanced semantic editor services as constraint problems. Constraints allow us to separate the declarative specification of a problem from the operational semantics needed to solve them. Currently, our constraint solver can only use the type system specification to verify the correctness of a program.

In this work we outline our ideas for extending the solver, such that it will be able to use the type system specification for advanced semantic editor services such as semantic code completion. We are currently implementing this extension to the Statix constraint solver, and evaluating its capabilities and performance.

```
typeOf(s, e) = ty :- e match {
  True() -> BOOL == ty.
  And(e1, e2) ->
    typeOf(s, e1) == BOOL,
    typeOf(s, e2) == BOOL,
    BOOL == ty.
  Call(x, es) ->
    Method{x} in s == METHOD(targs, tret),
    typesOf(s, es) == targs,
    tret == ty.
}
```

Figure 1. A single Statix typing rule for expressions, used to type check True, conjunction, and method calls.

2 Architecture

In the Spoofax language workbench [2] the static type system of a programming language can be specified using Statix, a meta-language for the declarative specification of static semantics [9]. It models the static semantics of a language as a constraint problem, and includes name binding and resolution by asserting structure and querying a *scope graph* [5, 8].

A typing rule for Java expressions is shown in Figure 1. Other rules would assert scopes and edges in the scope graph. Figure 2 shows a small Java example program and the corresponding scope graph, where scope 0 is the root scope in

```
public class C {
    boolean m(int x, int y) {
        return (x == y) && $Exp
    }

    int f() { return 42; }
}
```

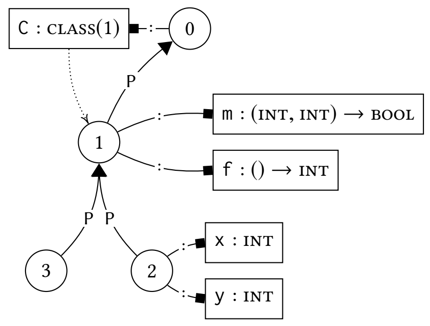

Figure 2. Incomplete example Java program with a placeholder $*Exp*, and its corresponding scope graph.

which the class C is declared, scope 1 is the scope within the class with the declarations of methods m and f, and scopes 2 and 3 are the bodies of those methods respectively.

The current implementation of the Statix solver is deterministic: it only applies a rule if it matches the given program construct, and does not do back-tracking. The algorithm uses the specification to simplify constraints until only core constraints remain, which are solved through unification and scope graph resolution.

We propose to extend the solver to include a limited form of search and back-tracking. The solver will perform a two-stage process: inference and search. In the first stage it will perform inference by simplifying and unifying constraints until it terminates or gets stuck. In the second stage a *search strategy* determines whether it performs a search by refining one of the constraints, splitting the search tree into multiple branches. The solver then loops and performs inference on each of the resulting branches. Branches that are not satisfiable are discarded. Other branches may yield a solution or get stuck, and the cycle repeats.

In Statix, syntax `match` constraints are refined into their constituent branches, and scope graph `query` constraints are refined into the various declarations that the queries could resolve to. The search strategy, which is specific to the editor service, determines which constraints are refined, and whether the algorithm continues the search. It employs these kinds of search:

- Non-deterministic — Refines a constraint
- Deterministic — Refines the only branch
- None — Performs simplification and unification only

Performing a non-deterministic search on a constraint variable will yield all solutions that are found after refining only one level deep. This is useful for semantic code completion, where we return each of the solutions as completion proposals to the user. Any constraint variables in the solution that are not ground are replaced by placeholders in the proposed syntax. For example, in Figure 2, invoking completion on the placeholder `$Exp` would suggest: `true`, `$Exp && $Exp`, and `m($Exps)`.

Deterministic search is useful to refine constraints for which there is only one possibility. For example, a deterministic search on the constraint variable representing the arguments to a method `m(`e`)` results in the exact number of arguments the method expects `m($Exp, $Exp)`. If instead we would have performed a non-deterministic search, we would also get all possible values for the method arguments.

3 Conclusion

Given a type system specification of a language, we believe many useful semantic editor services can be implemented in a language-parametric way by just varying the search strategy being used. This will allow complex editor services to be supported with little to no extra effort from the language developer. We are in the process of implementing and evaluating the new implementation of the Statix solver. This will also gain us insight into its limitations and which editor services can be specified using our techniques.

References

[1] Hendrik Bünder. 2019. Decoupling Language and Editor - The Impact of the Language Server Protocol on Textual Domain-Specific Languages. In *modelsward*. 129–140. https://doi.org/10.5220/0007556301290140

[2] Lennart C. L. Kats and Eelco Visser. 2010. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *OOPSLA*. 444–463. https://doi.org/10.1145/1869459.1869497

[3] Sven Keidel, Wulf Pfeiffer, and Sebastian Erdweg. 2016. The IDE portability problem and its solution in Monto. In *SLE*. 152–162.

[4] Linghui Luo, Julian Dolby, and Eric Bodden. 2019. MagpieBridge: A General Approach to Integrating Static Analyses into IDEs and Editors (Tool Insights Paper). In *ECOOP*. https://doi.org/10.4230/LIPIcs.ECOOP.2019.21

[5] Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. 2015. A Theory of Name Resolution. In *ESOP*. 205–231. https://doi.org/10.1007/978-3-662-46669-8_9

[6] Daniel A. A. Pelsmaeker. 2018. *Portable Editor Services.* Master's thesis.

[7] Daniel A. A. Pelsmaeker, Hendrik van Antwerpen, and Eelco Visser. 2019. Towards Language-Parametric Semantic Editor Services Based on Declarative Type System Specifications (Brave New Idea Paper). In *ECOOP*. https://doi.org/10.4230/LIPIcs.ECOOP.2019.26

[8] Hendrik van Antwerpen, Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. 2016. A constraint language for static semantic analysis based on scope graphs. In *PEPM*. 49–60. https://doi.org/10.1145/2847538.2847543

[9] Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. 2018. Scopes as types. *PACMPL* 2, OOPSLA (2018). https://doi.org/10.1145/3276484