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

Daniel A.A. Pelsmaeker
d.a.a.pelsmaeker@tudelft.nl

Hendrik van Antwerpen
h.vanantwerpen@tudelft.nl

Eelco Visser
e.visser@tudelft.nl

Overview

We propose to use constraint programming on syntax-directed typing rules to not only verify the correctness of the program, but also enable advanced semantic editor services.

By performing a search over the solution space, we get multiple possible solutions to a given constraint problem. We believe these can be used to implement language-parametric refactorings, code navigation, and semantic code completion with minimal effort.

Program

The user's program is incomplete. We represent holes in an incomplete AST with a syntactic placeholder and a corresponding constraint variable, which allows normal parsing and semantic analysis to take place.

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

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

Scope Graph

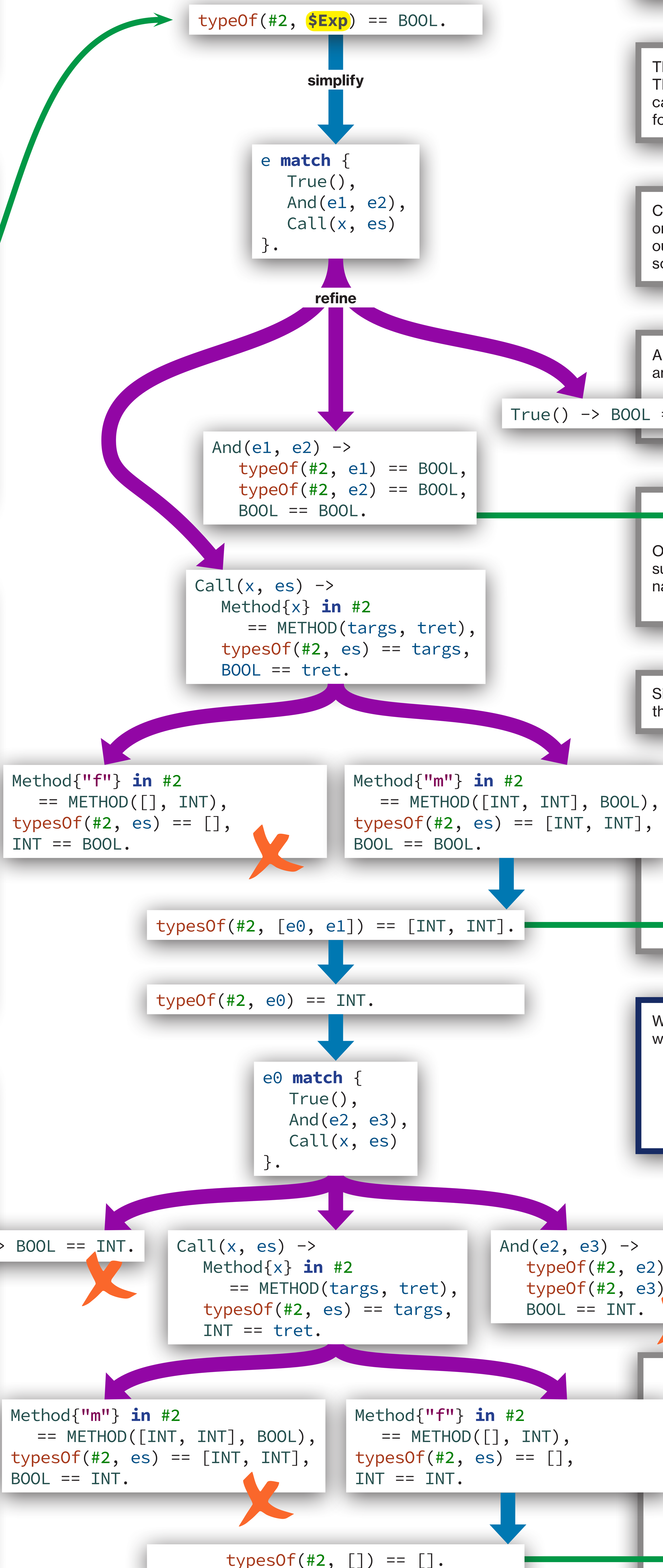
The scope graph is a representation of the declarations and scopes in the program. It is produced by the solver and used for name resolution.

C : CLASS(1) — 0; P; 1; m : INT, INT → BOOL; f : () → INT; P P; 3; 2; x : INT; y : INT

Static Semantic Rules

The static semantics of the language is expressed as declarative syntax-directed rules in Statix. Each rule specifies the constraints it applies to parts of the program, and how the scope graph should be extended.

```
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.

  // ...

}.
```



Semantic analysis on the incomplete program leaves us with some constraints on the placeholder. We use these as the starting point for our search algoritm. An editor service-specific *search strategy* dictates how the search is performed.

Through simplification we replace a constraint by its subconstraints. These constraints can be solved using unification. However, just simplification and unification will provide up to one solution, which makes it useful for program verification.

Constraint refinement splits a constraint in different constraints within the original constraint's domain, allowing us to explore different solutions. In our case we split the **match** constraint into its separate branches. Some solutions will not be satisfiable.

A solution can include an AST fragment. Their syntax can be used, for example, as a code completion proposal.

`true`

`$Exp && $Exp`

Our search strategy determines whether we continue our search for the subexpressions or terminate here with a partial solution. When we terminate, we replace the constraint variables by syntactic placeholders.

Similarly, we can perform refinement on a **query** constraint, refining it into the various declarations that would satisfy the query.

`m($Exps)`

The search strategy can decide to perform a deterministic search on some of the constraint variables. For a method call this would result in a syntax fragment with placeholders for the exact number of expected arguments.

`m($Exp, $Exp)`

We can use the solutions we found for semantic code completion, where each solution is a completion proposal:

```
return (x == y) && |
```

- `true`
- `$Exp && $Exp`
- `m($Exp, $Exp)`

If we continue our search, we may find a solution for the subexpressions as well. However, we run the risk of trying to find an infinite sequence of nested expressions.

`m(f($Exps), $Exp)`

The search strategy determines how we search and when we terminate.

`m(f(), $Exp)`

D. A. A. Pelsmaeker, H. van Antwerpen, E. Visser. (2019).
Towards Language-Parametric Semantic Editor Services based on Declarative Type System Specifications (Brave New Idea Paper).
In 33rd European Conference on Object-Oriented Programming (ECOOP 2019). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.