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

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, which allows normal parsing and semantic analysis to take place.

```
public class C {
    boolean n(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)
```

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 := -_match {
    True() -> BOOL => ty.
    And(e1, e2) ->
        ty1 = (s1, e1),
        ty2 = (s2, e2),
        ty = ty1 && ty2.
    Call(x, es) ->
        Method(x, es = METHOD(targs, tret),
        typesOf(s, es) = targs,
        tret = ty.
    // ...

    true
```

Method("M") in #2

```
== METHOD([], INT),
    typesOf(#2, es) = [],
    ty1 = INT
```

Method("M") in #2

```
== METHOD([INT, INT], BOOL),
    typesOf(#2, es) = [INT, INT],
    ty1 = BOOL
```

Method("M") in #2

```
== METHOD([INT, INT], BOOL),
    typesOf(#2, es) = [],
    ty1 = INT
```

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.

```
match { True(), And(e1, e2), Call(x, es) }.
```

```
typeOf(#2, $Exp$) = BOOL.
```

```
match { True(), And(e1, e2), Call(x, es) }.
```

```
typesOf(e1, e2, es) = [INT, INT],
```

```
typesOf(#2, es) = [INT, INT],
```

```
typesOf(#2, es) = [],
```

```
match { True(), And(e1, e2), Call(x, es) }.
```

```
typesOf(e1, e2, es) = [INT, INT],
```

```
typesOf(#2, es) = [INT, INT],
```

```
typesOf(#2, es) = [],
```

```
match { True(), And(e1, e2), Call(x, es) }.
```

```
typesOf(e1, e2, es) = [],
```

```
typesOf(#2, es) = [],
```

```
typesOf(#2, es) = [INT, INT],
```

```
match { True(), And(e1, e2), Call(x, es) }.
```

```
typesOf(e1, e2, es) = [],
```

```
typesOf(#2, es) = [INT, INT],
```

```
typesOf(#2, es) = [INT, INT],
```

```
match { True(), And(e1, e2), Call(x, es) }.
```

```
typesOf(e1, e2, es) = [INT, INT],
```

```
typesOf(#2, es) = [INT, INT],
```

```
typesOf(#2, es) = [],
```

```
match { True(), And(e1, e2), Call(x, es) }.
```

```
typesOf(e1, e2, es) = [INT, INT],
```

```
typesOf(#2, es) = [INT, INT],
```

```
typesOf(#2, es) = [],
```

```
match { True(), And(e1, e2), Call(x, es) }.