BUILDING TYPE CHECKERS USING SCOPE GRAPHS FOR A LANGUAGE WITH TYPE CLASSES

Responsible Professor & Supervisor
Casper Bach Poulsen

a.l.mocanu@student.tudelft.nl
Andreea Mocanu

Supervisor
Aron Zwaan

TU Delft

1 INTRODUCTION

Type checkers help developers catch errors in their code, such as type mismatches or undefined operations, early.

However, implementing type checkers is not a trivial task. One particular challenging aspect of type checkers is name binding (associating names with units).

Scope graphs provide a model for resolving names during type checking, uniformly and independently of language.

TERMINOLOGY

Scope graphs are graphs where each node represents a new scope.

A **type class** is a family of types that implement a common interface (set of functions).

An **instance** of the type class is a type that belongs to this family.

2 PURPOSE

"How can we build type checkers for languages with support for type classes, using scope graphs?"

"How does the declarativity and feature extensibility of the implemented type checker compare with the typing rules provided in [1]?"

3 LANGUAGE

The typechecker is implemented for a mini-language with support for type classes, with the following syntax:

```
data Type = NumT | BoolT        data Expr = Num Int | Bool Bool
 | FunT Type Type                  | Plus Expr Expr | Ident String
 | TyVar String                    | App Expr Expr | Abs String Expr
 | TyClass String                  | Let String Expr Expr

    data DeclT = ClassDecl String [Type] [DeclT]
         | InstDecl [Type] String [DeclT]
         | Method String Type Type
         | FunDecl String String DeclT Expr
```

4 METHOD

A Haskell library [3] is used to create scope graphs from programs.

Name resolution is resolved by finding a path from a reference to the corresponding declaration.

```
class A_1 a_2 where
 f_3 :: a_4 -> Bool

instance A_5 Int where
 f_6 :: Int -> Bool
 f_7 x_8 = True

foo_9 :: Int -> Bool
foo_10 x_11 = f_12 x_13
```

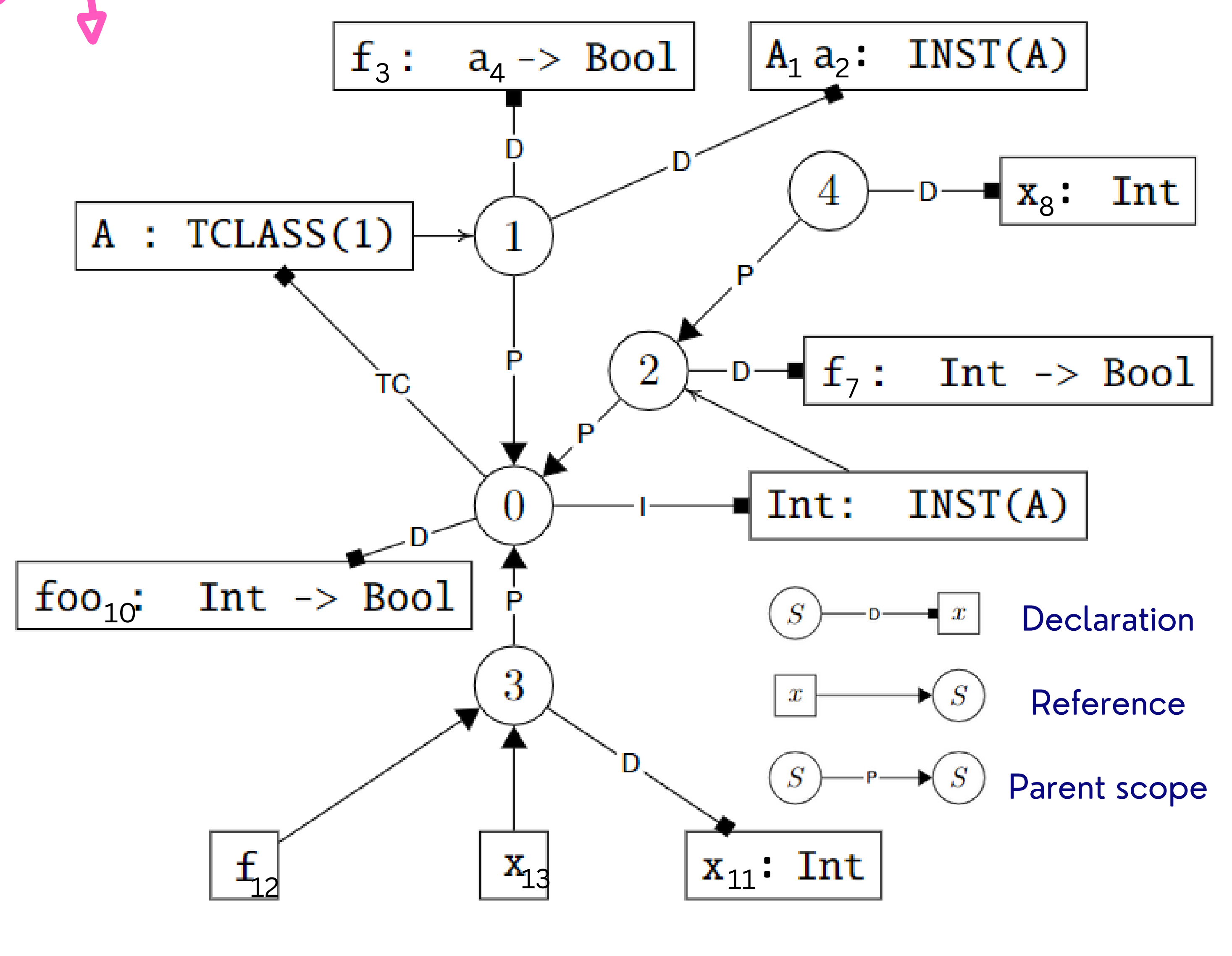
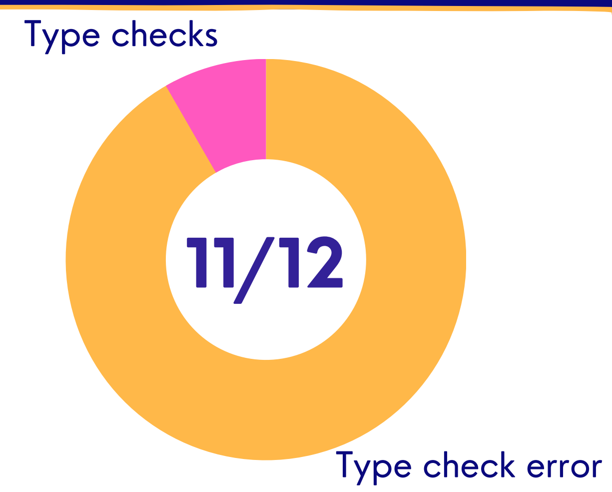


Fig 1: Scope graph for the code (left) and its legend (right).

5 RESULTS

The implementation provided [4] passes 30 out of 35 tests covering basic programs, instance resolution, overlapping instances. The test suite identified a bug when type checking functions declared within the scope of a type class or instance.

Compared to [1], the current implementation is intuitive and easy to understand, with the type checking algorithm having two distinct phases. However, it requires more type annotations than Haskell and does not support as many features. Current implementation offers comparable solutions for resolving qualified types (type constraints on type variables).



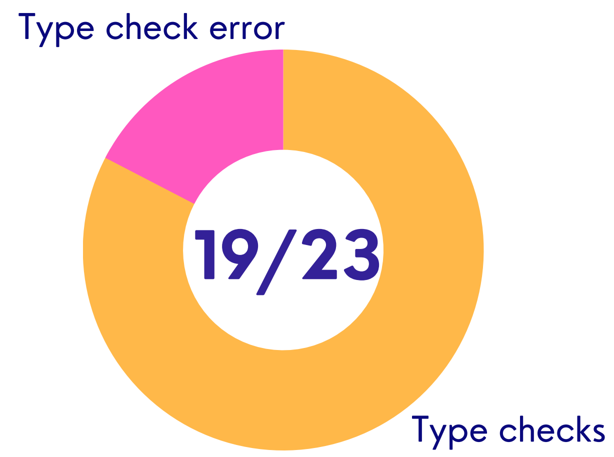


Fig 2: Two progress circles indicating the how many tests pass; programs that raise errors (top) and that type check (bottom).

6 CONCLUSION

The approach of using scope graphs is promising and intuitive.

Recommendations for future work:

- Add support for type constructors and subclasses to improve expressiveness and reusability, as well as provide more support for polymorphism.
- Implement a type inference algorithm such that the language requires less type annotations.

REFERENCES

[1] Mark P. Jones. "A System of Constructor Classes: Overloading and Implicit Higher-Order Polymorphism". In: Proceedings of the Conference on Functional Programming Languages and Computer Architecture. FPCA '93. Copenhagen, Denmark: Association for Computing Machinery, 1993, page 52-61. isbn: 089791595X. doi: 10 . 1145 / 165180 . 165190

[2] Robin Milner. A theory of type polymorphism in programming. Journal of computer and system sciences,17(3):348–375, 1978.

[3] https://github.com/heft-lang/hmg

[4] https://github.com/andreealmocanu/scope-graph-type-class