Building Type Checkers using Scope Graphs for a Language with a Substructural Type System

J.G. Knapen (j.g.knapen@student.tudelft.nl) - Supervisors: Casper Bach Poulsen, Aron Zwaan
CSE3000 27/6/2023

1. Background & Research Question

Type checker has ability to catch errors at compile time.

Substructural Type Systems
Linear types are used exactly once
Affine types are used at most once
Dependent types are dependent on other values/types

Scope graphs are a data structure that can represent lexical scoping of names in a program. They are directed graphs that capture the nesting of scopes and the relationships between them.

Monotonicity means that a fixed query in a fixed scope always returns the same result. This is an important feature of scope graphs.

Can we implement a type checker using scope graphs for languages with a substructural type system?

```
let x = 4
  in x + x
```

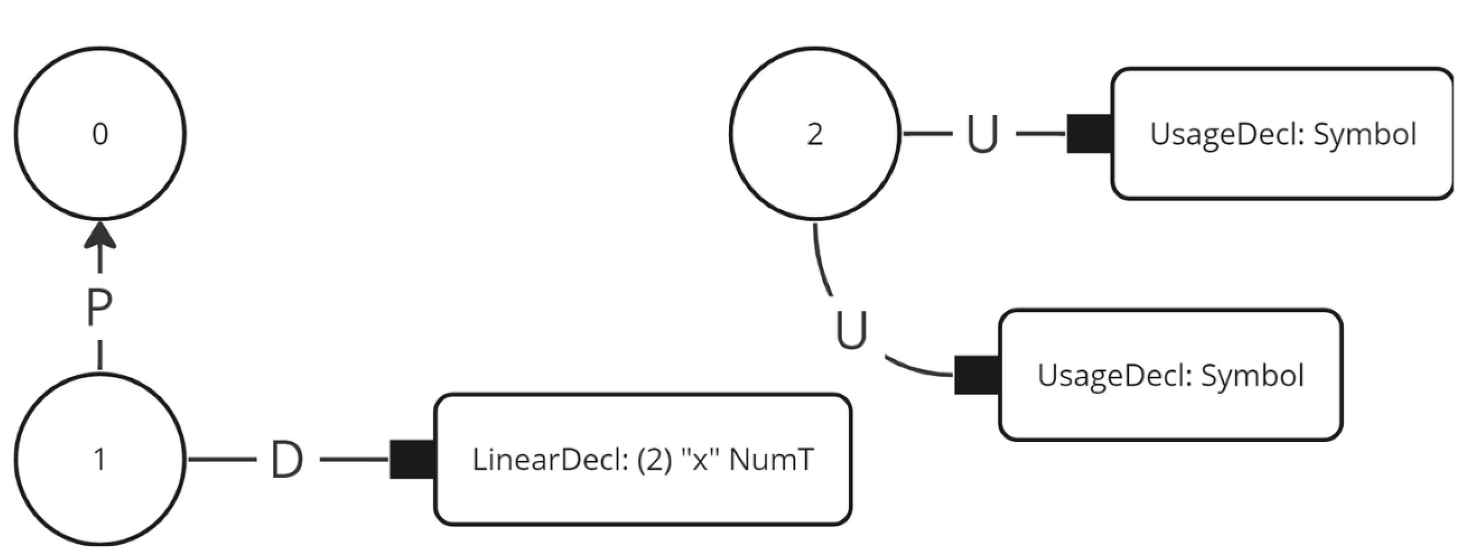

Fig. 1: Scope graph solution example

2. Problem Description

Phased Haskell library
1. ATerms
2. Abstract Syntax
3. Typechecking program

Methodology
1. (re)format typing rules of David (2002) [1]
2. Calculus implementation (without scope graphs)
3. Exploration of possible ideas and approaches
4. Implementation using scope graphs
5. Evaluation comparing both implementations

3. Contribution

Defined typing rules for following expressions:
- **Variable identifier:** Ident str
- **Addition:** Plus e1 e2
- **Application:** App e1 e2
- **Function Abstraction:** Abs str ty e1 e2
- **Let-Binding:** Let str ty e

Calculus implementation
- Used lang-hm [2] as boilerplate
- Extended type with linearT & affineT
- Added Let-Bindings
- Adjusted typecheck function to return context

Scope graph implementation
Solution: keep count of usages of a variable.
Added an extra phase at the end to check usage count of substructural types.

Syntax

```
data Expr
    = Num Int
    | Plus Expr Expr
    | App Expr Expr
    | Ident String
    | Abs String Type Expr
```

Type Checking

1. Input code is parsed to Expr
2. Expr is type checked while building scope graph
   a. UsageDecl is added to substructural variables each them they are queried
3. Substructural variables are checked using the UsageDecl count

4. Evaluation

Test suite
Some test cases different expected results due to phase differences between the two implementations.

Results

Test Suite	Cases	Tried	Failures
Non-substructural tests	41	41	0
Linear tests	37	37	6
Affine tests	37	37	6

calculus implementation

Test Suite	Cases	Tried	Failures
Non-substructural tests	41	41	0
Linear tests	37	37	0
Affine tests	37	37	0

scope graph implementation

Failures due to **Unification Error**, likely caused by recursive Abs.

Code analysis
- **Limited expressiveness:** no specific error for different substructural types
- **Good extensibility:** recursive substructural types and expressions extension require minimal knowledge
- **Limited documentation**

5. Conclusion and Future work

Conclusion
Successful implementation:
- Potentially easy extension with other substructural types
- Expressions extension with limited knowledge

Limited evaluation due to lack of comprehensive test suite.

Future Work
More comprehensive test suite:
- Cover additional edge cases
- Test cases with combinations of different typing

Explore integration of chosen solution with other language features & paradigms.

6. References

[1] David Walker. Substructural type systems. In https://mitpress-request.mit.edu/sites/default/files/titles /content/9780262 162289_sch_0001.pdf, page 10, 2002
[2] Jan Knapen. Scope graph scheduling bsc substructural type systems. https://github.com/JanKnapen/ scope-graph-scheduling-bsc-substructural. Accessed: June 23, 2023.