

Regras de Scope em Ztrategic

Dissertação de Mestrado



André Bernardo Coelho Nunes

Dissertação efetuada sob a orientação de:
João Alexandre Batista Vieira Saraiva

O que são Scopes e as suas regras?

- Um **Scope** refere-se à área em que uma função ou variável é **visível e acessível** a outro código, restringindo a visibilidade dos sítios de definição.
- **Regras de Scope** definem as **relações** entre definições e usos de nomes.

```
program = let  $a = b + 3$   
            $c = 2$   
            $b = c \times 3 - c$   
           in  $(a + 7) \times c$ 
```

Limitações das tecnologias e abordagens atuais

- Os **IDE's** atuais oferecem uma variedade de serviços quando se trata de resolução de nomes, mas todos **carecem de flexibilidade e eficiência para lidar com as nuances específicas de cada idioma.**
- Estes serviços são **desenvolvidos manualmente e feitos especificamente para cada idioma**, o que exige um trabalho substancial, tanto no seu desenvolvimento como na sua manutenção e evolução.
- Na maioria dos casos, os serviços para cada idioma são **pacotes independentes** mantidos por um determinado grupo de pessoas, o que significa que **existem vários grupos de pessoas a repetir o mesmo trabalho para diferentes idiomas.**

Motivação

- Linguagens reais são representadas por ASTs heterogêneas grandes e complexas.
- As regras de Scope dependem de algoritmos complexos de travessia múltipla.
- Linguagens diferentes usam regras de Scope ligeiramente diferentes.

É necessário um método simples, transparente e genérico o suficiente para analisar as particularidades de cada linguagem!

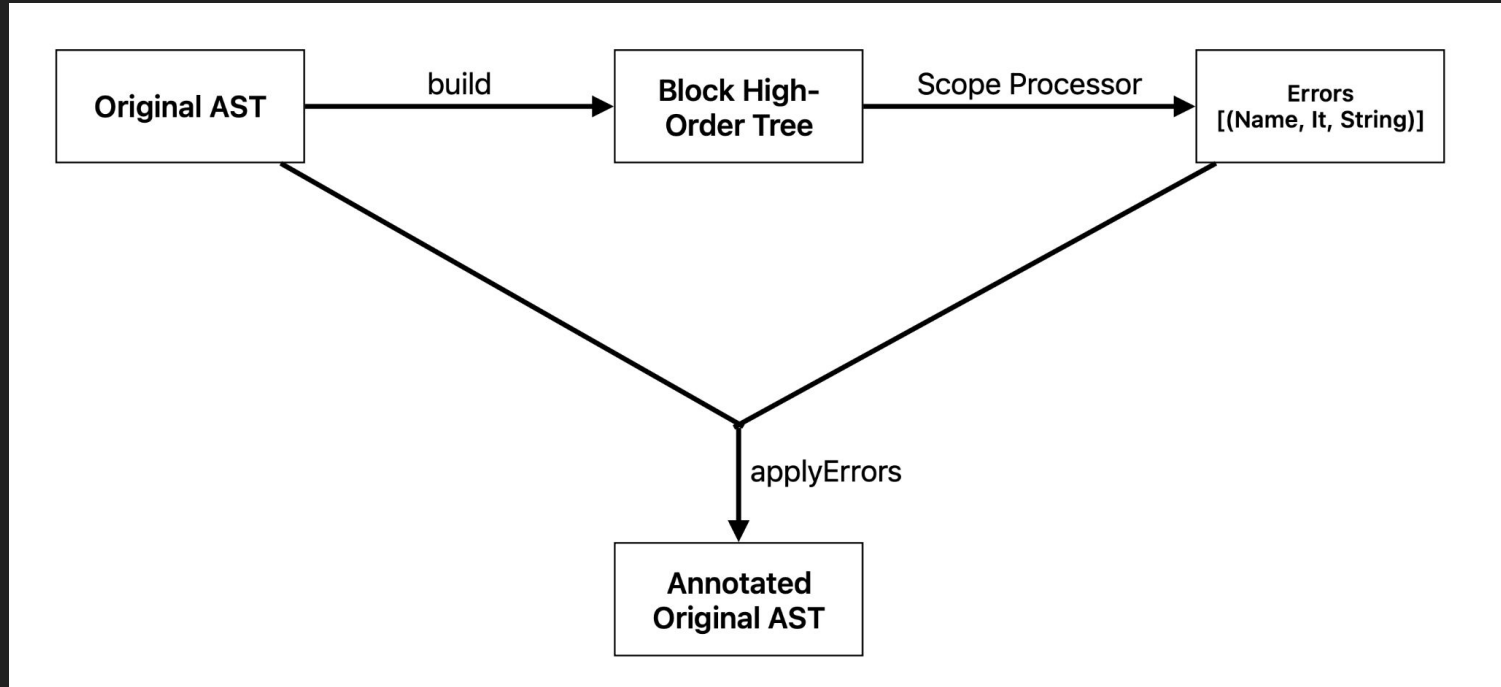
Uma solução possível é usar Gramáticas de Atributos de ordem superior. Mesmo que os erros detectados na AST de ordem superior não possam logo ser mapeados na AST original.

Questões de pesquisa

- É possível inferir **automaticamente** todo o processo de construção da representação onde expressamos a análise dos nomes?
- É possível definir um mecanismo que permita **signalizar erros na árvore original** e não naquele onde definimos as regras de nomes?
- É viável **fundir o funcionamento do processador da linguagem Block com o processo de análise de uma linguagem** para analisar nomes e regras de Scope **sem perder informações** sobre a linguagem em questão?

let w = b + -16	[decl w , use b
a = 8	, decl a
w = let z = a + b	, decl w , [decl z , use a , use b
in z + b	, use z , use b]
b = c + 3 - c	, decl b , use c , use c
in c + a - w	, use c , use a , use w]

A solução



Para a linguagem Let

```
data Root = Root Let
  deriving (Data, Typeable)

data Let = Let List Exp
  deriving (Data, Typeable)

data List
  = NestedLet Name Let List
  | Assign Name Exp List
  | EmptyList
  deriving (Data, Typeable)

data Exp = Add Exp Exp
  | Sub Exp Exp
  | Neg Exp
  | Var Name
  | Const Int
  deriving (Data, Typeable)

type Name = String
type Env = [(Name, Int, Maybe Exp)]
type Errors = [Name]
```



```
instance I.Scopes (L.Let) where
  isDecl ag = case (LS.constructor ag) of
    LS.CAssign -> True
    LS.CNestedLet -> True
    _ -> False
  isUse ag = case (LS.constructor ag) of
    LS.CVar -> True
    _ -> False
  isBlock ag = case (LS.constructor ag) of
    LS.CLet -> True
    _ -> False
  isGlobal ag = False
  initialState ag = []

instance StrategicData (L.Let) where
  isTerminal t = isJust (getHole t :: Maybe Int)
  || isJust (getHole t :: Maybe LS.Name)
```

let w = b + -16

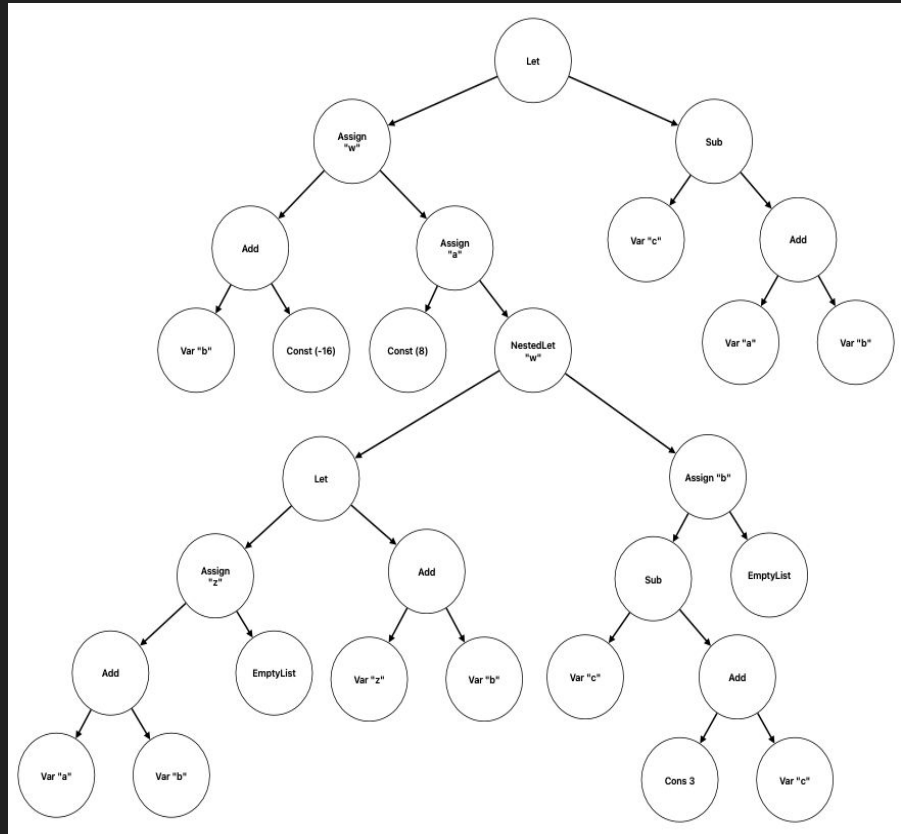
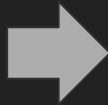
a = 8

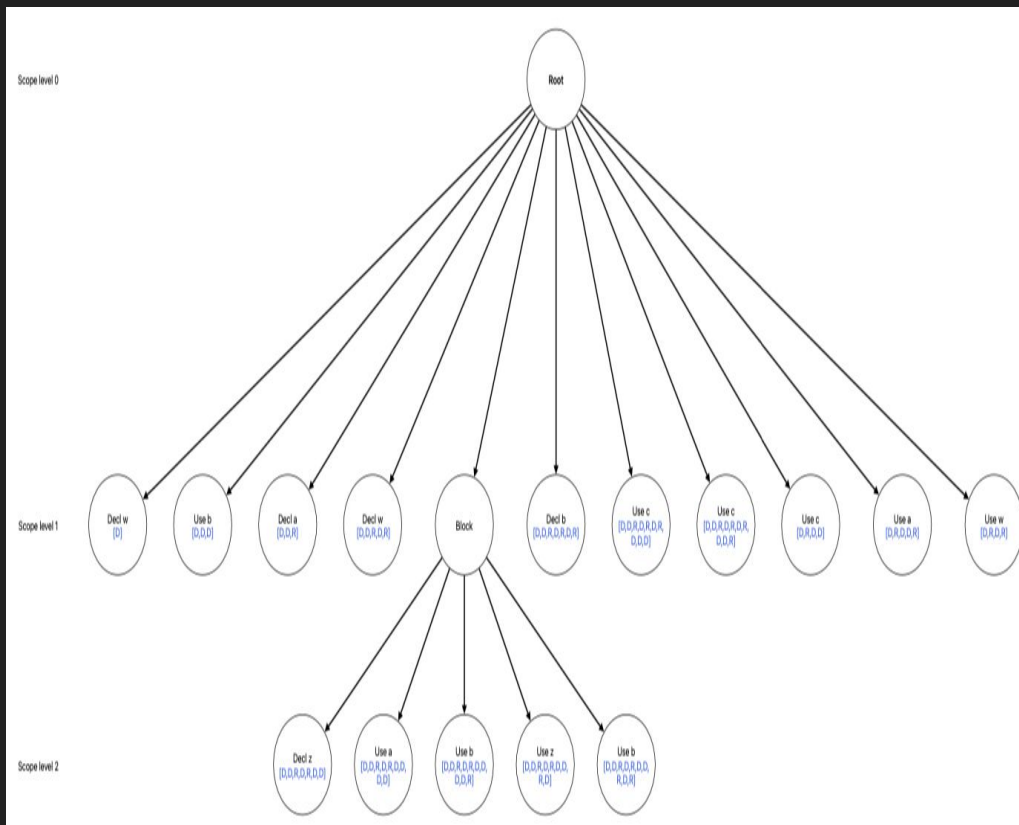
w = **let** z = a + b

in z + b

b = c + 3 - c

in c + a - w

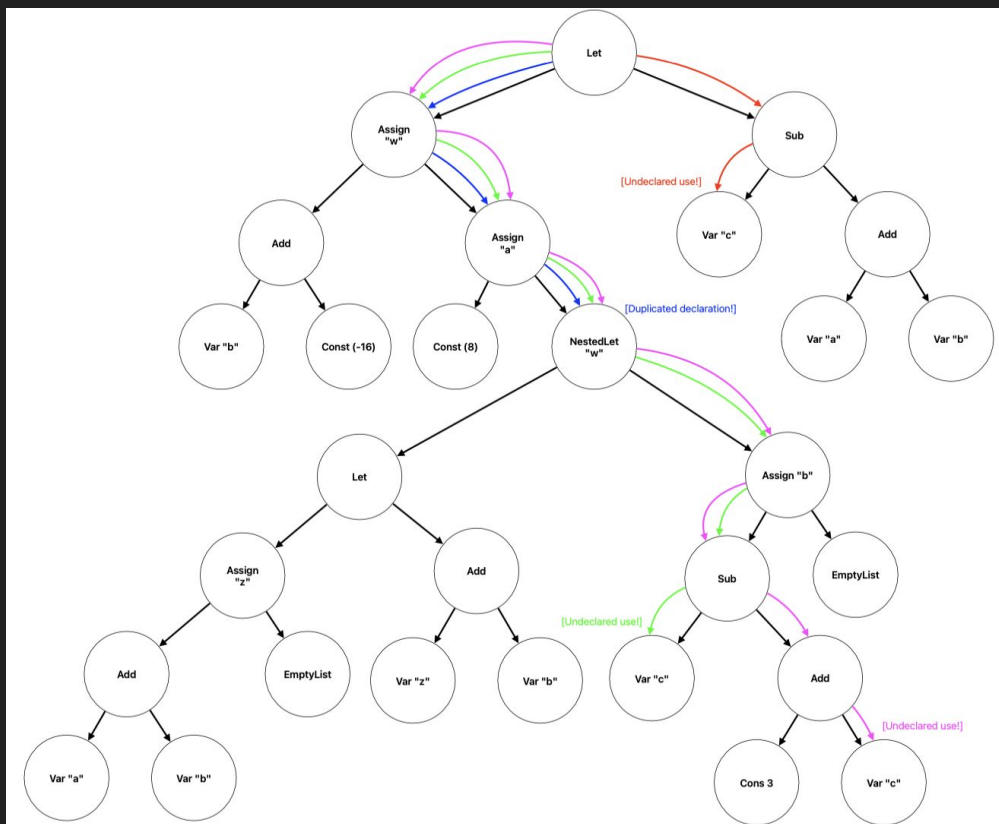




```

[ (Decl w | Path: [D]), (Use b | Path: [D,D,D]), (Decl a | Path: [D,D,R]),
  (Decl w | Path: [D,D,R,D,R]),
  [
    (Decl z | Path: [D,D,R,D,R,D,D]), (Use a | Path: [D,D,R,D,R,D,D,D,D]),
    (Use b | Path: [D,D,R,D,R,D,D,D,D,R]), (Use z | Path: [D,D,R,D,R,D,D,R,D]),
    (Use b | Path: [D,D,R,D,R,D,D,R,D,R]),
  ],
  (Decl b | Path: [D,D,R,D,R,D,R]), (Use c | Path: [D,D,R,D,R,D,R,D,D,D]),
  (Use c | Path: [D,D,R,D,R,D,R,D,D,R]), (Use c | Path: [D,R,D,D]), (Use a | Path: [D,R,D,D,R]),
  (Use w | Path: [D,R,D,R]) ]
  
```

Usando um qualquer
processador inferimos os
erros de **Scope** a partir
da **AST** de ordem
superior!



```
[("w", (Decl w | Path: [D,D,R,D,R]), " <= [Duplicated declaration!]"),
 ("c", (Use c | Path: [D,D,R,D,R,D,R,D,D,D]), " <= [Undeclared use!]"),
 ("c", (Use c | Path: [D,D,R,D,R,D,R,D,D,R]), " <= [Undeclared use!]"),
 ("c", (Use c | Path: [D,R,D,D]), " <= [Undeclared use!]")]
```

Com os erros encontrados podemos anotar a **AST original** visto que temos os caminhos para cada erro na mesma!

```
let w = b + -16
```

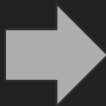
```
  a = 8
```

```
  w = let z = a + b
```

```
      in z + b
```

```
  b = c + 3 - c
```

```
in c + a - w
```



```
let w = b + -16
```

```
  a = 8
```

```
  w <= [Duplicated declaration!] = let z = a + b
```

```
      in z + b
```

```
  b = c <= [Undeclared use!] + 3 - c <= [Undeclared use!]
```

```
in c <= [Undeclared use!] + a - w
```

Quais as vantagens deste método?




- **Escalável**
 - O utilizador tem a habilidade de usar processadores pré-definidos (Algol 68 Rules, Object-Oriented Rules, Declare-Before-Use Rules) ou definir os seus.
- **Transparente**
 - O utilizador apenas tem que preencher a classe Scopes para tirar proveito de toda a interface.
 - Abstrai significativamente as complexidades das Regras de Scope.
- **Genérico**
 - O método pode ser usado em qualquer linguagem.

Resultados

- Extensão à biblioteca Zstrategic para análise de nomes automática.
- Paper submetido (em revisão) ao journal e conferência Programming'25.
- A extensão foi utilizada para implementar a análise de nomes para a linguagem Haskell, tal como é reportado no caso de estudo do paper.

Name Resolution for Free

A Zipper-based, Modular Embedding of Scope Rules

José Nuno Macedo^a , André Nunes^a, Marcos Viera^b , and João Saraiva^a 

^a Department of Informatics, University of Minho, Portugal

^b Instituto de Computación, Universidad de la República, Uruguay

The Art, Science, and Engineering of Programming

Perspective The Art of Programming

Area of Submission Social Coding, General-purpose programming



© José Nuno Macedo, André Nunes, Marcos Viera, and João Saraiva
This work is licensed under a “CC BY 4.0” license
Submitted to *The Art, Science, and Engineering of Programming*.

Conclusões

- Esta abordagem formal determina **automaticamente** o processo completo de criação de uma representação (AG de ordem superior) usada para analisar nomes e regras de Scope.
- Este formalismo faz parte de um conjunto mais amplo de funções que equipam a interface com capacidades para **verificar e anotar erros diretamente na AST original**.
- A abordagem demonstra que a combinação de técnicas de análise com o processador Block **garante análises precisas, transparentes e intuitivas sem perda de informações**.