

Analyse sémantique

L'analyse sémantique, consiste en une validation que la sémantique du langage est correcte. Quelques exemples:

- Variables lues avant initialisation
- Labels dupliqués dans un switch
- Réaffectation de constantes
- Visibilité des méthodes invoquées

Types de règles

- **Static rules:** vérifiées à la compilation
- **Dynamic rules:** vérifiées à l'exécution

Analyse de nom

L'analyse de nom consiste en la validation des identifiants des variables. On valide aussi le **scope** des variables. Les erreurs classiques analysées ici sont:

- **Nested scope:** les variables définies dans les scopes enfant ne sont pas accessible par les parents

```
int x = 10;
{
    int y = 20;
    // x et y accessibles ici
}
```

// seul x est accessible ici

- **Shadowing:** une variable dans une portée interne a le même nom qu'une variable dans une portée externe. La variable interne "masque" la variable externe.

- **Duplicated Definitions:** une variable est définie deux fois avec le même nom → erreur

Pour gérer l'analyse de nom on utilise une **tables des symboles** qui peut stocker les infos suivantes:

- Nom de la variable
- Type
- Portée
- Adresse mémoire (pour la génération de code)

Systèmes de types

- **Primitifs:** int, float, bool, char
- **Composés:** arrays, structs, classes
- **Fonction:** types des params. et retour

Strong vs Weak typing

- **Strong:** conversions de types explicites requises
- **Weak:** conversions implicites fréquentes

Inférence de types

Déduire automatiquement le type de la valeur:

```
var x = 42; // Int
```

```
var y = "hello"; // String
```

Type checking

Literals :

$$\begin{array}{l} \Gamma \vdash n : \text{int} \\ \Gamma \vdash \text{true} : \text{bool} \end{array}$$

Variables :

$$(x : \tau) \in \Gamma \implies \Gamma \vdash x : \tau$$

Addition :

$$\frac{}{\Gamma \vdash e_1 + e_2 : \text{int}}$$

Augmenter l'environnement:

$$\text{LETIN } \frac{\Gamma \vdash e_1 : T_1 \quad \Gamma[x \rightarrow T_1] \vdash e_2 : T_2}{\Gamma \text{ let } x = e_1 \text{ in } e_2 : T_2}$$

Règles de fonctions

Application :

$$\frac{\Gamma \vdash f : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e : \tau_1}{\Gamma \vdash f(e) : \tau_2}$$

Si f prend un T_1 et retourne un T_2 et que e est un T_1 , alors $f(e)$ donne un T_2 .

Abstraction :

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2}$$

Si en supposant que x est de type T_1 , l'expression e a le type T_2 , alors la fonction $\lambda x. e$ a le type $T_1 \rightarrow T_2$.

$\lambda x. e$: fonction anonyme x param, e corps de la fonction.

Arbres de dérivation

Un arbre de dérivation est la preuve visuelle qu'un programme respecte les règles de types, construite en empilant les règles.

$$\begin{array}{c} \text{Lit } \Gamma \vdash i : \text{int} \qquad \text{IDENT } \frac{\Gamma(x) = T}{\Gamma \vdash x : T} \qquad \text{ADD } \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}} \\ \text{PROD } \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 * e_2 : \text{int}} \\ \\ \text{IDENT } \frac{\Gamma(z) = \text{int}}{\Gamma \vdash z : \text{int}} \qquad \text{LIT } \frac{\Gamma \vdash 1 : \text{int}}{\Gamma \vdash z+1 : \text{int}} \\ \text{PROD } \frac{\Gamma \vdash 2 : \text{int} \quad \text{ADD } \frac{\Gamma \vdash z : \text{int}}{\Gamma \vdash z+1 : \text{int}}}{\Gamma \vdash 2 * (z+1) : \text{int}} \end{array}$$

Erreurs de typage courantes :

- Type mismatch : $\text{int } x = \text{"hello"}$
- Fonction appelée avec mauvais arguments
- Opération non supportée : "hello" - 5
- Retour de type incorrect

Inférence de type

Mécanisme pour déterminer les types d'expressions à la compilation **sans annotations de type explicites**.

```
-- Type inféré: id :: a -> a
id x = x
a = id True      -- a :: Bool
b = id "hello"   -- b :: String
```

Type Variables

Placeholders pour types pas encore connus :

```
a, b, T
```

```
swap :: (a, b) -> (b, a)
```

```
swap (x, y) = (y, x)
```

Type Constraints

Conditions imposées sur les types basées sur leur usage.

Formes :

1. **Equality Constraints** : Types identiques requis

2. **Function Type Constraints** : Contraintes sur paramètres et retour

Unification

Résout les contraintes de type en trouvant itérativement des substitutions pour les variables de type.

Algorithm Hindley–Milner

Algorithm largement utilisé pour l'inférence de type.

Étapes :

1. **Type Variables** : Assigner variables de type uniques
2. **Type Constraints** : Générer contraintes basées sur opérations
3. **Unification** : Trouver substitution cohérente
4. **Generalization** : Trouver le type le plus général

Statements et états

Un **statement** est une instruction qui effectue une action, comme l'affectation d'une variable ou une boucle.

• **Statements:** modifient l'état du programme.

• **Expressions:** produisent des valeurs.

Environnement

L'environnement est une structure de données qui mappe les variables à leurs valeurs actuelles dans le contexte d'exécution. Les fonctions d'évaluation utilisent l'environnement pour accéder et modifier les variables.

type Env = Map String Value

Compilateurs

Gestion mémoire

Interpréteurs

Les interpréteurs se basent sur les étapes d'analyse lexicale, syntaxique et sémantique pour traduire du code source en actions exécutables.

Les langages interprétés sont souvent **plus faciles à apprendre, plus rapide à utiliser, indépendant de la plateforme et plus facile à débugger**. Cependant, ils peuvent être **plus lents** en termes de performance, **moins optimisés et moins adaptés** aux applications nécessitant une haute performance.

REPL

- **Read** : Lit l'entrée de l'utilisateur.
- **Eval** : Évalue l'expression ou la commande.
- **Print** : Affiche le résultat de l'évaluation.
- **Loop** : Répète le processus pour chaque nouvelle entrée.

AST (Abstract Syntax Tree)

Représentation arborescente abstraite de la structure syntaxique du code source.



Vaudrait l'expression: $2 + 3 * 4$, en Haskell nous pouvons écrire la fonction pour évaluer l'AST:

```
eval :: Expr -> Int
eval (Const n) = n
eval (Binary left op right) =
  case op of
    '+' -> eval left + eval right
    '-' -> eval left - eval right
    '*' -> eval left * eval right
    '-' -> error "Unknown operator"
```