idea
基础软件中心

语义分析与类型(Part1)

基础软件理论与实践公开课

ZhangYu

Theory study

high-level language

machine instruction

```
type rec lambda =
  Var(string)
| Fn(string, lambda)
| App(lambda, lambda)
```

STLC

System F

```
type rec expr =
  Cst(int)
| Add(expr, expr)
| Mul(expr, expr)
```

compile

```
type instr =
  Cst(int) | Add | Mul
```

introduce local variables

```
type rec expr =
| ...
| Let(string, expr, expr)
| Var(string)
```

```
type rec expr =
| ...
| Let(expr, expr)
| Var(int)
```

compile

```
type instr =
| ...
| Var(int) | Pop | Swap
```

introduce c-style functions

```
type rec expr =
| ...
| Letfn(string,
  list<string>, expr, expr)
| App(string, expr)
```

compile

```
type instr = ...
| Label(l)
| Call(l,n) | Ret(n)
| Goto(l) | IfZero(l)
```

first-class functions

```
type rec expr =
  Cst(int)
| Prim(prim, list<expr>)
| Let(string, expr, expr)
| Var(string)
| Fn(list<string>, expr)
| App(expr, expr)
```

TODO

let-polymorphism

```
type rec expr = ...
type rec ty = ...
```

idea 基础软件中心

Overview: Semantics analysis

idea
基础软件中心

- Semantics analysis
  - Name resolution (scope analysis)
  - Type system
- Today's class:
  - Name resolution
  - Simply-typed lambda calculus (STLC)
  - System F
- Next class:
  - Let-polymorphism
  - Implementing HM type system

What is semantics analysis?

For example:

- Will this function terminate on some input?
- Will this function ever use a variable not in the environment?
- Will this function treat a string as a function?
- Will this function divide by zero

Name resolution (scope analysis)

Motivation:

- Flatten the nested scope
- Enable/simplify program transformation/optimization
- Complex scope analysis is coupled with type system

Example

- Incorrect transformation:

```
let x = (1, 2) in
switch x {
  (x, y) => ...
}
```

is naively transformed to

```
let x = (1, 2) in
let x = fst x in
let y = snd x in
  ...
```

- how to make it correct?

Example

```
let x = 1 in
let x =
  let x = 2 in
  let y = x + x in
    y
in x + x
```

after name resolution and flattening:

```
let x/1 = 1 in
let x/2 = 2 in
let y/3 = x/2 + x/2 in
let x/4 = y/3 in
  x/4 + x/4
```

Implementation

idea
基础软件中心

```
type rec expr =
  | Cst(int) | Var(string)
  | Let(string, expr, expr) | Prim(prim, list<expr>)

type ident = { name: string, stamp: int}

module Resolve = {
  type rec expr =
  | Cst(int) | Var(ident)
  | Let(ident, expr, expr) | Prim(prim, list<expr>)
}

let fresh = (x: string): ident => { ... }
```

- More efficient than string representation -- only need to compare the stamp
- More friendly to program transformation/optimization than de Bruijn index

Implementation

```
let resolve = expr => {
  let rec go = (env: list<(string, ident)>, expr: expr): Resolve.expr =>
    switch expr {
    | Cst(i) => Cst(i)
    | Var(x) => Var(List.assoc(x, env))
    | Let(x, e1, e2) => {
        let fresh_x = fresh(x)
        Let(fresh_x, go(env, e1), go(list{(x, fresh_x), ...env}, e2))
      }
    | Prim(op, es) => Prim(op, List.map(go(env), es))
  }
  go(list{}, expr)
}
```

- Partial evaluation

Type

What is a type?

> A concise, formal description of the behavior of a program fragment

Why type?

> Well-typed programs do not go wrong

- Detecting errors as early as possible
- Abstraction -- type driven development
- Documentation
- Help IDE refactor code
- ...

Simply typed lambda calculus

- Types

$$T ::= \mathsf{Int} \mid \mathsf{Bool} \mid T \to T$$

```
type rec ty = TInt | TBool | TArr (ty, ty)
```

- Terms

$$t ::= i \mid b \mid x \mid \lambda x : T . t \mid t\ t$$

```
type rec t = CstI (int) | CstB (bool) | Var (string)
           | App (t, t) | Abs (string, ty, t)
```

Runtime semantics

- Type-passing semantics

$$\frac{t_1 \to t_1'}{t_1\ t_2 \to t_1'\ t_2} \qquad \frac{t_2 \to t_2'}{v_1\ t_2 \to v_1\ t_2'} \qquad \frac{}{(\lambda x : T.\, t)\ v \to t[v/x]}$$

- Type-erasing semantics
  - $\lfloor \lambda x : T.\, t \rfloor = \lambda x.\, t$
  - overloading vs. dynamic dispatch
- Could be used to generate more efficient programs

Typing

- Typing environment

$$\begin{aligned}\Gamma ::= & \\ & \mid \epsilon \\ & \mid \Gamma,\ x : T\end{aligned}$$

- Typing rules

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\text{T-Var} \qquad \frac{}{\Gamma \vdash i : \mathsf{Int}}\text{T-Int} \qquad \frac{}{\Gamma \vdash b : \mathsf{Bool}}\text{T-Bool}$$

Typing

- Typing rules

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \to T_2} \text{T-Abs}$$

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2} \text{T-App}$$

Implementation

idea
基础软件中心

- Syntax-directed

```
type tenv = list<string, ty>
let rec typeof = (ctx, expr) => {
  switch expr {
  | CstB(_) => TBool | CstI(_) => TInt
  | Var(x) => List.assoc(x, ctx)
  | Abs(x, t, body) => {
      let bodyTy = typeof(list{(x,t), ...ctx}, body)
      TArr(t, bodyTy)
    }
  | App(rator, rand) => {
      let TyArr(paramTy, resTy) = typeof(ctx, rator) // report errors
      assert (paramTy == typeof(ctx, rand)); // FIXME: report errors
        resTy
    }
  }
}
```

Example

type check

$$(+) : \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} \vdash \lambda x : \mathsf{Int}.\ x + 3$$

Type safety

- What is the meaning of "Well-typed programs do not go wrong", formally?

If $\vdash t : T$ and $t \to^* t'$ and t' cannot be further reduced, then t' is a value and $\vdash t' : T$

- Preservation

If $\vdash t : T$ and $t \to t'$, then $\vdash t' : T$

- Progress

If $\vdash t : T$, then either t is a value or there exists an t' such that $t \to t'$

Type inference

For example, the type annotation of $\lambda x : \mathsf{Int}.\ x + 3$ is redundant

Type inference: constraint solving

For example, we want to infer the type T in

$$\vdash \lambda x.\, \lambda y.\, \lambda z.\, (x\ z)\ (y\ z) : T$$

- Insert type variables

$$\vdash \lambda x : X.\, \lambda y : Y.\, \lambda z : Z.\, (x\ z)\ (y\ z) : T$$

- Generate constaints
  - $(x\ z)\ (y\ z) : T$ means $(x\ z) : T_1 \to T$ and $(y\ z) : T_1$, where T_1 is fresh
  - $x\ z : T_1 \to T$ means $X = T_2 \to (T_1 \to T)$ and $Z = T_2$, where T_2 is fresh
  - $y\ z : T_1$ means $Y = T_3 \to T_1$ and $Z = T_3$, where T_3 is fresh

Type inference: constraint solving

- Overall, the constaints we have collected

$$
\begin{aligned}
X &= T_2 \to (T_1 \to T) \\
Y &= T_3 \to T_1 \\
Z &= T_2 \\
Z &= T_3
\end{aligned}
$$

- Solving the constraints by hand

$$
\begin{aligned}
\vdash & \lambda x : T_2 \to (T_1 \to T). \\
& \lambda y : T_2 \to T_1. \\
& \lambda z : T_2. \\
& (x\ z)\ (y\ z) : T
\end{aligned}
$$

What's the problem with STLC?

- Limited expresive power
  - Can you type $\omega = \lambda x.\, x\ x$?
- More seriously

```
let id = fun x -> x in
  (id(1), id(true))
```

- Duplication vs. abstraction

```
let id_int = ...
let id_bool = ...
```

System F (Polymorphic lambda calculus)

- Intuition

$$\mathsf{id} = \lambda X : \mathrm{Type}.\, \lambda x : X.\, x$$
$$\mathsf{id} : \forall X.\, X \to X$$

- Instantiation

$$\mathsf{id}_{\mathsf{int}} = \mathsf{id}\ [\mathsf{Int}] = \lambda x : \mathsf{Int}.\, x$$
$$\mathsf{id}_{\mathsf{int}} = \mathsf{Int} \to \mathsf{Int}$$
$$\mathsf{id}_{\mathsf{bool}} = \mathsf{id}\ [\mathsf{Bool}] = \lambda x : \mathsf{Bool}.\, x$$
$$\mathsf{id}_{\mathsf{bool}} = \mathsf{Bool} \to \mathsf{Bool}$$

System F (Polymorphic lambda calculus)

- Syntax

$$t ::= i \mid b \mid x \mid \lambda x : T.\, t \mid t\ t \mid \lambda X : \mathrm{Type}.\, t \mid t\ [T]$$

- Types

$$T ::= \mathsf{Int} \mid \mathsf{Bool} \mid T \to T \mid X \mid \forall X.\, T$$

For example, in Haskell, all type variables are universally quantified by default

```
id :: a -> a
id x = x
```

System F

- Type check: $\lambda X : \mathrm{Type}.\, t$

$$\frac{\Gamma, X : \mathrm{Type} \vdash t : T}{\Gamma \vdash \lambda X : \mathrm{Type}.\, t : \forall X.\, T}\text{T-TAbs}$$

- Type check: $t\ [T_1]$

$$\frac{\Gamma \vdash t : \forall X.\, T}{\Gamma \vdash t\ [T_1] : T[T_1/X]}\text{T-TApp}$$

System F: examples

- twice and fourtimes

$$\begin{aligned}\mathsf{twice} &= \lambda X.\, \lambda f : X \to X.\, \lambda a : X.\, f\ (f\ a)\\ \mathsf{twice} &: \forall X.\, (X \to X) \to X \to X\\ \mathsf{fourtimes} &= \lambda X.\, \mathsf{double}\ [X \to X]\ (\mathsf{double}\ [X])\\ \mathsf{fourtimes} &: \forall X.\, (X \to X) \to X \to X\end{aligned}$$

- self application

$$\omega = \lambda x : \forall X.\, X \to X.\, x\ [\forall X.\, X \to X]\ x$$

- what is the type of ω?

Decidability

- Type inference for System F is undecidable
- Soundness, completeness and decidability
- Sweet spot: let-polymorphism

Recommended reading and references

[1] Section 8-10, 22, 23 in Types and Programming Languages

[2] Type systems for programming languages, Didier Rémy

[3] Programming Languages, Dan Grossman

Let-polymorphism

```
let twice = fun f -> fun a -> f(f(a)) in
let x = twice (fun x -> x + 1) 3 in
let y = twice (fun x -> not x) true in
...
```

- type variables range only over quantifier-free types
- quantified types are not allowed to appear on the left-hand sides of arrow types