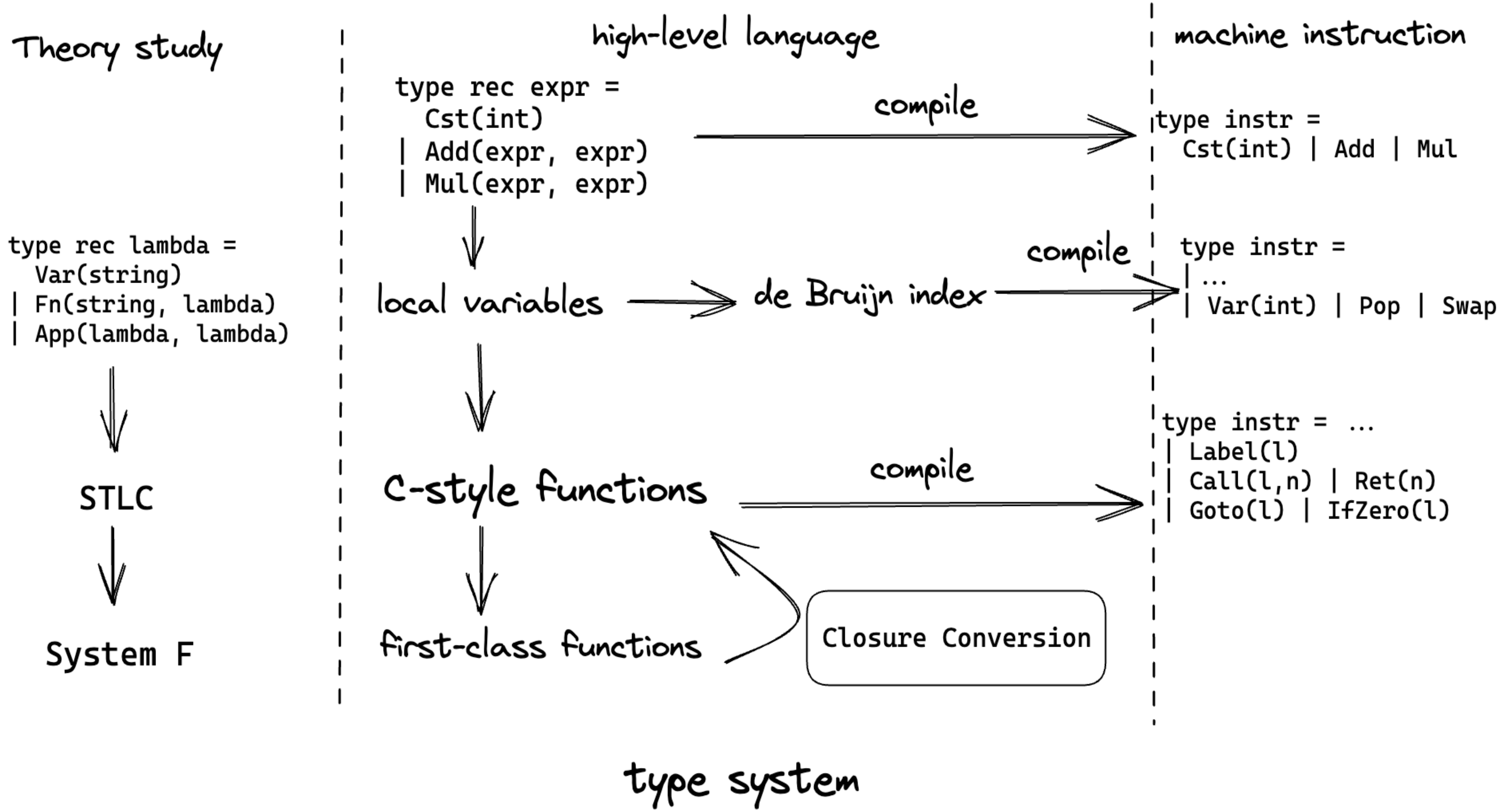
idea
基础软件中心

闭包的表示与编译(Part1)

基础软件理论与实践公开课

ZhangYu

Roadmap


Theory study
high-level language
machine instruction
type rec expr =
Cst(int)
| Add(expr, expr)
| Mul(expr, expr)
compile
type instr =
Cst(int) | Add | Mul
type rec lambda =
Var(string)
| Fn(string, lambda)
| App(lambda, lambda)
local variables
de Bruijn index
compile
type instr =
| ...
| Var(int) | Pop | Swap
STLC
C-style functions
compile
type instr = ...
| Label(l)
| Call(l,n) | Ret(n)
| Goto(l) | IfZero(l)
System F
first-class functions
Closure Conversion
type system

Contents

Recap

We will use the language from Lec 2:

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

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

Introduction

- We have shown the compilation for C-like functions
  - no nested functions
  - no free variables: neither local variables nor parameters of the function
  - each function can be simply represented as a function pointer
  - use the `call` instruction to jump to the label for the function
- Goal: compile **first-class** functions to C-like functions

Introduction

Considering compiling the following code:

```
let make_adder(x) =
  let inner_func(y) = x + y in
  inner_func

let add2 = make_adder(2) in
add2(3)                              // supposed to return 2+3=5
```

- Functions can be used as return value
- Inner function may **capture** variables, such as `x`

which means, local defined function may have longer lifetime than its parent

- `inner_func` lives longer than `make_adder`
- but `inner_func` refer to the binding `x` defined in `make_adder`

Introduction

Can we **directly** flatten the functions?

```
let inner_func(y) = x + y          // where is x?????

let make_adder(x) = inner_func
```

`inner_func` accesses to a *free variable*!

To deal with this situation, we need **closure conversion**.

Intuition

Recap:

- We used **closures** to implement the interpreter for first-class functions
  - A function pointer and an environment (for interpreting free variables)
- Interpreters create closures in the host language
- Compilers make the closures explicit in the compiled program

Intuition:

- Using closures to transform the program with first-class functions to C-like functions

Intuition

We first use closures to eliminate free variable `x`

```
let make_adder(x) =
  let inner_func(y) = x + y in
  inner_func

let add2 = make_adder(2) in
add2(3)
```

To make `inner_func` closed, we introduce an environment as an extra parameter

```
let make_adder(x) =
  let inner_func(env, y) = env.x + y in  // note the extra parameter env
  (inner_func, new_env({ x := x }))       // return the closure

let add2 = make_adder(2) in
add2(3)                                   // program breaks here
```

基础软件中心

Intuition

The closure creation and application should conform to a protocol

```
let make_adder(x) =
  let inner_func(env, y) = env.x + y in
  (inner_func, new_env({ x := x }))      // closure creation

let add2_clo = make_adder(2) in
let (add2_func, add2_env) = add2_clo in  // decompose the closure
add2_func(add2_env, 3)                   // pass the env to the function
```

Now the program should work

Intuition

Finally, we can lift the nested function to **toplevel** (aka hoisting).

Example:

```
let inner_func(env, y) = env.x + y
let make_adder(x) = (inner_func, new_env({x := x}))

let add2_clo = make_adder(2) in
let (add2_func, add2_env) = add2_clo in
add2_func(add2_env, 3)
```

Nice! Now we can proceed with the compilation scheme from Lec 4.

idea
基础软件中心

Summary

- Remove free variables by closures conversion
  - Closure creation
  - Closure application
- Hoist the functions to top level

Formalization

Recap: by studying lambda calculus, we can focus on the most essential part

We can transform our program to lambda calculus, for example:

$$
\begin{aligned}
\texttt{let}\ \ f(x) = a\ \ \texttt{in}\ \ b \qquad & \text{transforms to} \qquad \texttt{let}\ \ f = \lambda x.\, a\ \ \texttt{in}\ \ b \\
\texttt{let}\ \ x = e\ \ \texttt{in}\ \ b \qquad & \text{transforms to} \qquad (\lambda x.\, b)\ e
\end{aligned}
$$

Formalization: free variables

The set of free variables can be inductively defined as follows:

$$\begin{aligned} \mathtt{fv}(x) &= \{x\} \\ \mathtt{fv}(e_1\ e_2) &= \mathtt{fv}(e_1) \cup \mathtt{fv}(e_2) \\ \mathtt{fv}(\lambda x.\, e) &= \mathtt{fv}(e) \setminus \{x\} \end{aligned}$$

For example,

$$\mathtt{fv}(\lambda x.\, y + x) = \{y\}$$

Formalization: representation of closures

Closure is represented as a tuple that contains:

- function pointer
- captured variables.

$$\text{closure} \equiv (f, (x_1, \ldots, x_n))$$

where x_i are the free variables in the function `f`.

Note there are different ways to represent the closures

Formalization: closure conversion

Finally, we show the inductive definition of closure conversion:

$$
\begin{aligned}
[\![x]\!] &= x \\
[\![\lambda x.\, t]\!] &= \texttt{let}\ f = \lambda(x, (x_1, \ldots, x_n)).\, [\![t]\!]\ \texttt{in} \\
&\quad\ (f, (x_1, \ldots, x_n)) \\
[\![t_1\ t_2]\!] &= \texttt{let}\ clo = [\![t_1]\!]\ \texttt{in} \\
&\quad\ \texttt{let}\ f = fst(clo)\ \texttt{in} \\
&\quad\ \texttt{let}\ env = snd(clo)\ \texttt{in} \\
&\quad\ f\ ([\![t_2]\!], env)
\end{aligned}
$$

where x is the original function parameter, and $\{x_1, \ldots, x_n\} = \texttt{fv}(\lambda x.\, t)$.

Note the conversion is closely related to the closure representation

Discussion

For another example,

```
let map(f, xs) =
  let go(xs) = match xs with
  | [] -> [] | x :: xs -> f(x) :: go(xs)
  in go(xs)
in
let scale(k, xs) =
  map (fun x -> k * x) xs
in scale(2, [1,2,3])
```

We can transform `fun x -> k * x` to a closure

But how about the recursive function `go`?

and how do we compile the call `f(x)` where `f` is a variable?

and ...

idea
基础软件中心

Problems to think about

- How to free the allocated closures?
- How to identify function calls which don't need to be transformed?
- How to handle recursive function?
- Indirect call

Q&A