Reinvent λ calculus by yourself

基础软件理论与实践公开课

张宏波

Review

- Compile $\mathsf{Nameless.expr}$ to machine instructions
- Introducing yet another IR

```
module Indexed = {
    type rec expr =
    | Cst (int)
    | Add (expr,expr)
    | Mul (expr,expr)
    | Var ({bind : int, stack_index: int}) // compute the stack_index
    | Let (expr, expr)
}
```

Review

```
type sv = Slocal | Stmp
type senv = list<sv>

let compile = (expr) => {
  let rec go = (expr: Nameless.expr, senv: senv) : Indexed.expr => {
    switch expr {
      | Cst(i) => Cst(i)
      | Var(s) =>  Var({bind:s, stack_index : sindex(senv, s)}) // calculate the index of Slocal
      | Add(e1, e2) =>  Add(go(e1, senv), go(e2, list{Stmp,... senv})
      | Mul(e1, e2) =>  Mul(go(e1, senv), go(e2, list{Stmp,... senv}))
      | Let(e1, e2) =>  Let(go(e1, senv), go(e2, list{Slocal,... senv}))
    }
  }
  go(expr, list{})
}
```

Review

Now linearize the indexed IR.

```
let compile = (expr) => {
     let rec go = (expr: Indexed.expr) : list<instr> => {
      switch expr {
        | Cst(i) => list{ Cst(i) }
        | Var({stack_index,_}) => list{ Var(stack_index) }
        | Add(e1, e2) => concatMany([ go(e1), go(e2), list{ Add } ])
        | Mul(e1, e2) => concatMany([ go(e1), go(e2), list{ Mul } ])
        | Let(e1, e2) => concatMany([ go(e1), go(e2}), list{ Swap, Pop } ])
      }
    }
    go(expr, list{})
}
```

Review

- Merge the two passes in a single pass

```
type sv = Slocal | Stmp
type senv = list<sv>

let scompile = (expr) => {
  let rec go = (expr: Nameless.expr, senv: senv) : list<instr> => {
    switch expr {
      | Cst(i) => list{ Cst(i) }
      | Var(s) => list{ Var(sindex(senv, s)) } // calculate the index of Slocal
      | Add(e1, e2) => concatMany([ go(e1, senv), go(e2, list{Stmp,... senv}), list{ Add } ])
      | Mul(e1, e2) => concatMany([ go(e1, senv), go(e2, list{Stmp,... senv}), list{ Mul } ])
      | Let(e1, e2) => concatMany([ go(e1, senv), go(e2, list{Slocal,... senv}), list{ Swap, Pop } ])
    }
  }
  go(expr, list{})
}
```

Review

- Dedicated syntax sugar in the next version of ReScript

```
type sv = Slocal | Stmp
type senv = list<sv>

let scompile = (expr) => {
  let rec go = (expr: Nameless.expr, senv: senv) : list<instr> => {
    switch expr {
      | Cst(i) => list{ Cst(i) }
      | Var(s) => list{ Var(sindex(senv, s)) }
      | Add(e1, e2) => list{ ...go(e1, senv), ...go(e2, list{Stmp,... senv},  Add }
      | Mul(e1, e2) => list{ ...go(e1, senv), ...go(e2, list{Stmp,... senv}), Mul }
      | Let(e1, e2) => list{ ...go(e1, senv), ...go(e2, list{Slocal,... senv}),  Swap, Pop }
    }
  }
  go(expr, list{})
}
```

Review

- `expr` -> `Nameless.expr` -> `Indexed.expr` --> linearize --> Stackmachine
- Merge three passes a single pass

```
  type sv = Slocal(string) | Stmp
  type senv = list<sv>

  let scompile = (expr) => {
    let rec go = (expr: expr, senv: senv) : list<instr> => {
      switch expr {
        | Cst(i) => list{ Cst(i) }
        | Var(s) => list{ Var(sindex(senv, s)) }
        | Add(e1, e2) => concatMany([ go(e1, senv), go(e2, list{Stmp,... senv}), list{ Add } ])
        | Mul(e1, e2) => concatMany([ go(e1, senv), go(e2, list{Stmp,... senv}), list{ Mul } ])
        | Let(x, e1, e2) => concatMany([go(e1, senv), go(e2, list{Slocal(x),... senv}), list{ Swap, Pop } ])
      }
    }
    go(expr, list{})
  }
}
```

Tiny language 3

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

- Tiny language 3 is turing complete

Abstraction

- Design: divide the programs into smaller pieces
- Explicit inputs and outputs
- Functions in *math*, functional
    - Same input, same output
    - Easy to debug, reason

Eval of tiny language 3

- What is the value of evaluating a function ??
- The function could capture state!

Interpreter

idea 基础软件中心

```
type rec value =
   | Vint (int)
   | Vclosure (env, list<string>, expr)
and env = list<(string, value)>
let vadd = (v1, v2) : value => { ... } // type error when add int and closure
let vmul = (v1, v2) : value => { ... }

let rec eval = (expr : expr, env : env) : value => {
  switch expr {
   | Cst (i) => Vint(i)
   | Add(a,b) => vadd(eval (a, env), eval (b, env))
   | Mul(a,b) => vmul(eval (a, env), eval (b, env))
   | Var(x) => List.assoc (x, env)
   | Let(x,e1,e2) => eval(e2, list{(x, eval(e1, env)), ...env})
   | Fn(xs, e) => Vclosure(env, xs, e) // computation suspended for application
   | App(e, es) => {
       let Vclosure(env_closure, xs, body) = eval(e, env)
       let vs = map(es,  e => eval(e, env))
       let fun_env = concatMany( [zip (xs, vs) , env_closure] )
       eval(body, fun_env)
     }
   }
}
```

Tiny language 4 : Nameless style

idea
基础软件中心

```
type rec expr =
  ...
  | Fn (expr) // no need to store the arity, precompute the index of parameters
  | App (expr, list<expr>) // we need semantics checkig!

type rec value =
  | Vint (int)
  | Vclosure (env, expr)
and env = list<value>
let rec eval = (expr, env) : value => {
  switch expr {
  ...
  | Fn(e) => Vclosure(env, e)
  | App(e, es) => {
      let Vclosure(env_closure, body) = eval(e, env)
      let vs = map(es, e => eval(e, env))
      let fun_env = concatMany( [vs, env_closure] ) // piece together
      eval(body, fun_env)
    }
  }
}
```

idea
基础软件中心

What's the problem of our interpreter ??

Simplify the language

Why?

Simplification to lambda calculus

- With functions and applications, the Let-expression turns out to be unnecessary

$$\mathsf{Let}(x, e_1, e_2) \equiv \mathsf{App}((\mathsf{Fn}(x, e_2)), e_1)$$

- Not necessary in theoretic study, but needed for practical reaons: performance

Simplification of primitives

- `Cst i` -> `Church numerals`
- `Bool`

Currying

- We can transform a function with multiple arguments to the form where it accepts the first argument and returns a function that accepts the second argument and so on.

$$\begin{aligned} \mathsf{Fn}(x_1, x_2, e) &\equiv \mathsf{Fn}(x_1, \mathsf{Fn}(x_2, e)) \\ \mathsf{App}(e, e_1, e_2) &\equiv \mathsf{App}(\mathsf{App}(e, e_1), e_2) \end{aligned}$$

You could reinvent the Lambda calculus

- The simplified language looks like

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

- "Calculus" just means a bunch of rules for manipulating symbols

Why study lambda calculus

- Small core, easy to make formal proof

> By the way, why did Church choose the notation "λ"? In [A. Church, 7 July 1964. Unpublished letter to Harald Dickson, §2] he stated clearly that it came from the notation "$\hat{x}$" used for class-abstraction by Whitehead and Russell, by first modifying "$\hat{x}$" to "$\wedge x$" to distinguish function-abstraction from class-abstraction, and then changing "$\wedge$" to "λ" for ease of printing. This origin was also reported in [J. B. Rosser. Highlights of the history of the lambda calculus. Annals of the History of Computing, 6:337—349, 1984, p.338]. On the other hand, in his later years Church told two enquirers that the choice was more accidental: a symbol was needed and "λ" just happened to be chosen.

idea
基础软件中心

Formally

Assume given an infinite set $\mathcal{V}$ of variables, denoted by x, y, z, etc. The set of lambda terms Λ is given by

$$\text{Lambda terms:} \qquad M, N ::= x \mid (MN) \mid (\lambda x.\, M)$$

The following are some examples of lambda terms:

$$(\lambda x.\, x) \qquad ((\lambda x.\, (xx))(\lambda y.\, (yy))) \qquad (\lambda f.\, (\lambda x.\, (f(fx))))$$

```
type rec t =
    | Var (string)
    | App (t,t)
    | Fun (string, t)
```