Adv. Course in Programming Languages

Yukiyoshi Kameyama

Department of Computer Science, University of Tsukuba

How to *stage* programs, precisely

Today's Topic

- Type system of programming languages
- Type inference
- Type inference for staging (based on examples)

Type System

```
let f x y z = if x then y + 1 else z * 3
==>
f : bool -> int -> int -> int
```

This type means

- x has type `bool`
- y has type `int`
- z has type `int`
- then, the return type of f is `int`

Benefit of Types

Basic benefits

- Avoiding illegal instructions (e.g. `10 + "abc"`), "well-typed programs don't go wrong".
- Lightweight documentation
- Efficiency (i.e. hint for compiler)
- Abstraction (interface vs implementation)

Advanced benefits

- Types can include more static information
- E.g. Static/Dynamic (staging), Invariant (verification), Security etc.

Types for functional programming languages

```
let foo x   = x+1   : int->int

let goo x y = x+y : int->int->int

let hoo x y = if x then y else y + 1
                    : bool->int->int
```


Type inference rules

e = 13

- type of e must be int

e = e1 + e2

- type of e1 must be int
- type of e2 must be int
- type of e must be int

Type inference rules

e = `if e1 then e2 else e3`

- type of e1 must be bool
- type of e2 can be arbitrary type (α)
- type of e3 must be α
- type of e must be α

e = `x`

- type of `x` can be arbitrary type (α)
- type of e must be α
- (and all occurrences of `x` must have the same type)

Example of type inference

```
let rec power n x =
  if n=0 then 1
  else x * (power (n-1) x)
```

- let α be type of `n`
- let β be type of `x`
- let γ be return type of `power`
- `n=0` has type `bool`, α must be `int`
- `1` has type `int`, so γ is `int`
- `n-1` has type `int`
- `(power (n-1) x)` has type γ,
- `x*(power (n-1) x)` has type `int` and β is `int`
- and everything else is OK

Hence, `power` has type `int->int->int`.

Types for staging (naive version)

Type `S`

- means type for static values (will be computed and erased)
- e.g. the parameter n of power has type `S`

Type `D`

- means type for dynamic values (will become code)
- e.g. the parameter x of power has type `D`
- e.g. the return type of power is `D`

Type $t1 \rightarrow t2$

- means function type from $t1$ to $t2$.
- e.g. power is expected to have type `S -> D -> D`

No types such as `int`, `bool->bool` etc.

Notes on Types for Staging

Type `D` and type `D -> D` differ:

A MetaOCaml expression of Type `D`

```
<fun x -> x * 3>
```

A MetaOCaml expression of Type `D->D`

```
fun x -> < ~x * 3>
```

A MetaOCaml expression of Type `S->D`

```
fun x -> if x then <3> else <5>
```


Typing rules for staging

e = `e1 + e2`

- type of `e1`, `e2`, and `e` are `S`, or
- type of `e1`, `e2`, and `e` are `D`

e = `if e1 then e2 else e3`

- type of `e1` must be `S` or `D`, and
- type of `e2,e3,e` must be the same
- if type of `e1` is `D`, so is `e`

Typing rules for staging

e = `e1 e2` (function call)

- type of `e1`, `e2`, `e` are `D`, or
- type of `e1` is $\alpha \rightarrow \beta$, where α is type of `e2` and β is type of `e`

e = c (constant)

- type of `e` is `S` or `D`

Example (1)

Power function:

```
let rec power n x =
  if n=0 then 1
  else x * (power (n-1) x)
```

Suppose `n : S` and `x : D`, return type is `D`.

- `power : S -> D -> D`
- `(n=0) : S` and `(n-1) : S`
- `1 : D` or `1 : S`
- `(power (n-1) x): D`
- `x*(power (n-1) x): D`, hence then-part also has type `D`.
- `if ...then...else... : D` (the same as the return type)
- Ok!

Example (2)

Gib function:

```
let rec gib n x y =
  if n=0 then (x,y)
  else
  let (x1,y1)=gib (n-1) x y in
    (y1, x1+y1)
```

Suppose `n : S`, `x : S`, `y:D` and return type is unknown (α).

- `gib: S -> S -> D ->` α.
- `(n=0) : D` or `(n=0) : S`.
- `(x,y) : S*D` and α`=S*D`.
- `(gib (n-1) x y): S*D`.
- `x1 : S` and `y1 : D`.
- `(x1+y1)` is NOT typable

Refined system

Static values can be lifted to dynamic values.

```
lift 5
==> <5>

lift 5 : int code
lift   : int -> int code, or S -> D
```

When we stage a program we are allowed to insert lift at any place.

Example (2, again)

Gib function:

```
let rec gib n x y =
  if n=0 then (x,y)
  else
  let (x1,y1)=gib (n-1) x y in
    (y1, x1+y1)
```

Suppose `n : S`, `x : S`, `y:D` and return type is unknown (α).

- `gib: S -> S -> D -> \alpha`.
- `(x,y) : S*D` and α`=S*D`.
- `(gib (n-1) x y): S*D`, and `x1 : S` and `y1 : D`.
- `(lift(x1)+y1) : D`.
- `(x,y)` in then-branch should be `(lift(x),y)` of type `D * D`.
- Ok! gib has type `S -> S -> D -> (D * D)`.

Exercise: Infer the types

(1) Assume $x : D$, $y : S$, and return type is D.

```
let rec goo x y =
  if x > 100 then x − y
  else goo (goo (x+11) y) y
```

(2) Assume a, b, c have type S and d and e have type D. return type is D.

```
let rec foo a b c d e =
  if a > 0 then
   foo (a+b) (b+c) (c+d) (d−e) (e∗a)
  else
   foo 3 5 (foo b c d e a) 7 9
```


Summary

Type system and type inference:

- Key component in modern programming languages (C, C++, ML family, Haskell, Java, Scala, etc.)
- Important tool for program analysis and verification; we can represent various static information, including binding time (static/dynamic), invariant, security level, by types.
- A very simple type system help stage programs.