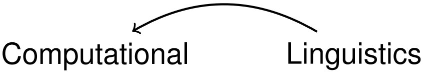
Quotation and effects in natural language

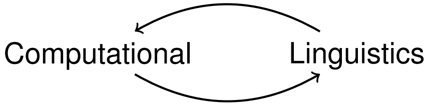
Three applications

Chung-chieh Shan   Oleg Kiselyov

21 August 2007

Computational Linguistics


Computational
Linguistics


Computational
Linguistics

?

Outline

- **Past: Mixed quotation**

Present: Quantifier scope
- Quotation for programming: code generation
- Control for programming: let insertion
- Control for linguistics: quantification
- Quotation for linguistics: inverse scope

Future: Rational metaprogramming

Anaphora as state

*Apparently, the idea of **meeting participants** making **their** own reservations at the hotel does not work well for them.*

Mixed quotation

“Bachelor” has eight letters.

Mixed quotation

"Bachelor" has eight letters.

Direct

Quine said, "Quotation has a certain anomalous feature".

Indirect

Quine said that quotation has a certain anomalous feature.

Mixed

Quine said that quotation "has a certain anomalous feature".

Mixed quotation

“Bachelor” has eight letters.

Direct

Quine said, “Quotation has a certain anomalous feature”.

Indirect

Quine said that quotation has a certain anomalous feature.

Mixed

Quine said that quotation “has a certain anomalous feature”.
Bush said he has an “ecelectic” reading list.
Bush said the enemy “misunderestimates me”.

Anaphora in quotation

The professor said she requires "every student in my class who lives on campus" to bring their homework to her office.

Professor to journalist:

> *I require every student in my class who lives on campus to drop their work into this box.*

Run with state?

Anaphora in quotation

The professor said she requires "every student in my class who lives on campus" to bring their homework to her office.

Professor to journalist:

> *I require every student in my class who lives on campus to drop their work into this box.*

Run with state?

* *The professor told every student in her class who lives on campus to "bring their homework to my office".*

Professor to John:

> *Please bring your Lordship's homework to my office.*

Professor to Mary:

> *Please bring your Ladyship's homework to my office.*

No cross-stage persistence?

Outline

Past: Mixed quotation

- **Present: Quantifier scope**
  - Quotation for programming: code generation
  - Control for programming: let insertion
  - Control for linguistics: quantification
  - Quotation for linguistics: inverse scope

Future: Rational metaprogramming

Staging power

```
let rec power1 x = function
    | 0 -> 1
    | n -> x * power1 x (pred n)
▶ val power1: int -> int -> int = <fun>

let test1 = power1 2 11
▶ val test1: int = 2048
```

Staging power

```
let rec power1 x = function
    | 0 -> 1
    | n -> x * power1 x (pred n)
▶ val power1: int -> int -> int = <fun>

let test1 = power1 2 11
▶ val test1: int = 2048


let rec power2 x = function
    | 0 -> ⟨1⟩
    | n -> ⟨~x * ~(power2 x (pred n))⟩
▶ val power2: ('a,int) code -> int -> ('a,int) code = <fun>

let test2 = ⟨fun x -> ~(power2 ⟨x⟩ 11)⟩
▶ val test2: ('a,int->int) code = ⟨fun x -> x*(x*(x*(x*(x*(
```

Interpreting English

`John loves Mary`

- `-: bool = true`

`John loves himself`

- `-: bool = false`

`Someone loves John`

- `-: bool = true`

Interpreting English

```
John loves Mary
```
▶ `-: bool = true`

```
John loves himself
```
▶ `-: bool = false`

```
Someone loves John
```
▶ `-: bool = true`

```
John loves Mary
```
▶ `-: ('a,bool) code = ⟨love Mary John⟩`

```
John loves himself
```
▶ `-: ('a,bool) code = ⟨love John John⟩`

```
Someone loves John
```
▶ `-: ('a,bool) code = ⟨List.exists people (love John)⟩`

The need to insert let

```
let square3 x =    x *  x

let rec power3 x = function
    | 0 ->  1
    | 1 -> x
    | n when n mod 2 = 0 -> power3 (square3 x) (n/2)
    | n ->    power3 (square3 x) (n/2)  *  x

let test3 =             power3  2  11
▶ val test3: int = 2048
```

The need to insert let

```
let square4 x = ⟨~x * ~x⟩

let rec power4 x = function
    | 0 -> ⟨1⟩
    | 1 -> x
    | n when n mod 2 = 0 -> power4 (square4 x) (n/2)
    | n -> ⟨~(power4 (square4 x) (n/2)) * ~x⟩

let test4 = ⟨fun x -> ~(power4 ⟨x⟩ 11)⟩
▶ val test4: ('a, int -> int) code
  = ⟨fun x -> (((x*x)*(x*x))*((x*x)*(x*x)))*(x*x)*x⟩
```

Inserting let in continuation-passing (or monadic) style

```
let square4 x   = ⟨         ~x * ~x              ⟩
```

Inserting let in continuation-passing (or monadic) style

```
let square5 x k = ⟨let r = ~x * ~x in ~(k ⟨r⟩)⟩
```

Inserting let in continuation-passing (or monadic) style

```
let square5 x k = ⟨let r = ~x * ~x in ~(k ⟨r⟩)⟩

let rec power5 x k = function
    | 0 -> k ⟨1⟩
    | 1 -> k x
    | n when n mod 2 = 0
      -> square5 x (fun s -> power5 s k (n/2))
    | n -> square5 x (fun s -> power5 s (fun r ->
                                  k ⟨~r * ~x⟩)
                                (n/2))

let test5 = ⟨fun x -> ~(power5 ⟨x⟩ (fun r -> r) 11)⟩
▶ val test5: ('a, int -> int) code
  = ⟨fun x -> let r1 = x  * x  in
              let r2 = r1 * r1 in
              let r3 = r2 * r2 in (r3 * r1) * x⟩
```

Inserting let in direct style

```
let square6 x = shift (fun k -> ⟨let r = ~x * ~x in ~(k ⟨r⟩)⟩)

let rec power6 x = function
    | 0 -> ⟨1⟩
    | 1 -> x
    | n when n mod 2 = 0 -> power6 (square6 x) (n/2)
    | n -> ⟨~(power6 (square6 x) (n/2)) * ~x⟩
```

```
let test6 = ⟨fun x -> ~(reset (fun () -> power6 ⟨x⟩ 11))⟩
▶ val test6: ('a, int -> int) code
  = ⟨fun x -> let r1 = x  * x  in
              let r2 = r1 * r1 in
              let r3 = r2 * r2 in (r3 * r1) * x⟩
```

Outline

Past: Mixed quotation

- **Present: Quantifier scope**
  - Quotation for programming: code generation
  - Control for programming: let insertion
  - Control for linguistics: quantification
  - Quotation for linguistics: inverse scope

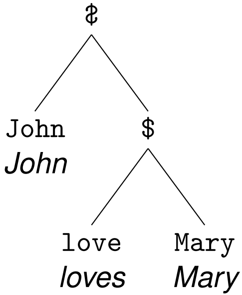
Future: Rational metaprogramming

Shifting gears

```
type person = John | Mary
let people = [John; Mary]
let love (x: person) (y: person) = x != y
let f $ x = f x
let x $ f = f x
```

`John $ (love $ Mary)`

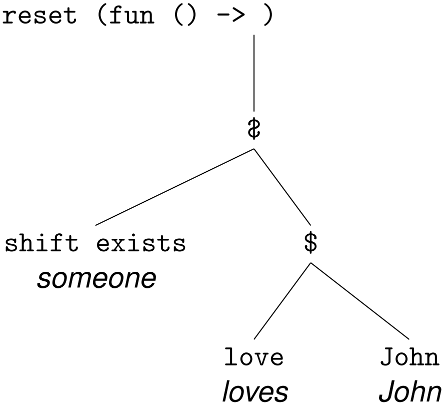
John loves Mary.



In-situ quantification

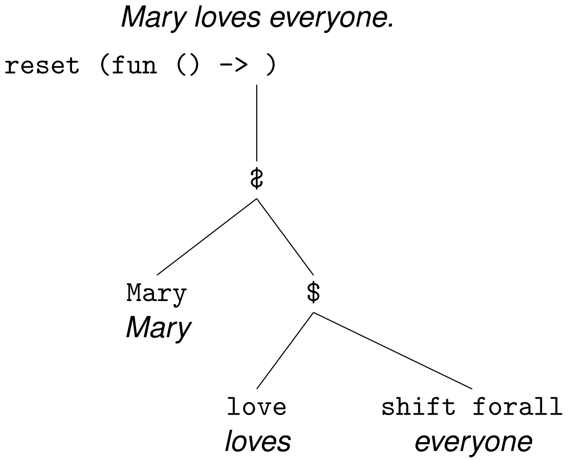
```
let forall (f: person -> bool) = List.for_all f people
let exists (f: person -> bool) = List.exists  f people
```

Someone loves John.



In-situ quantification

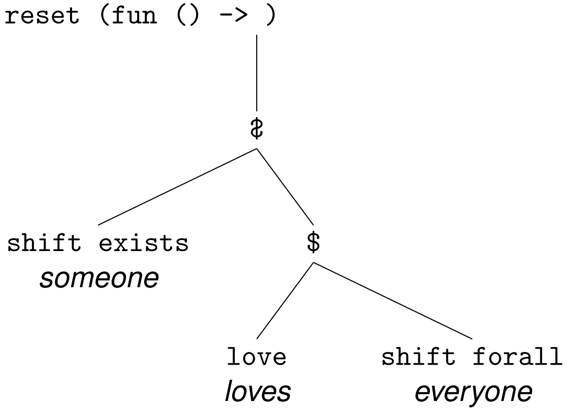
```
let forall (f: person -> bool) = List.for_all f people
let exists (f: person -> bool) = List.exists  f people
```



In-situ quantification

```
let forall (f: person -> bool) = List.for_all f people
let exists (f: person -> bool) = List.exists  f people
```

Someone loves everyone.



Scope ambiguity

Someone loves everyone.

Scope ambiguity

Someone loves everyone.

Children. . .
There's a time and a place for everything

Scope ambiguity

Someone loves everyone.

Children. . .
There's a time and a place for everything,
and it's called college.

Scope ambiguity

Someone loves everyone.

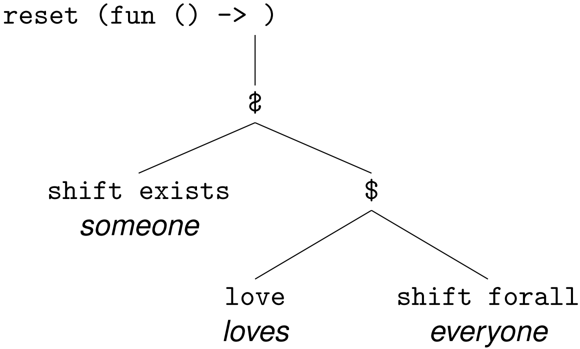
Children. . .
There's a time and a place for everything,
and it's called college.

Require left-to-right evaluation for other side effects:

* *His mother loves everyone.*
* *What did who buy?*
* *Anyone loves no one.*

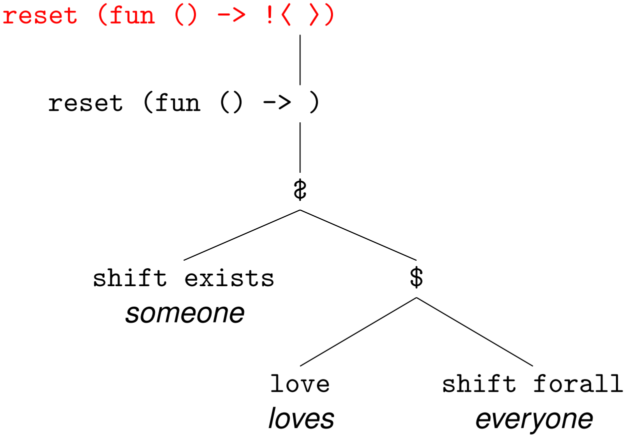
Inverse scope as quotation

Someone loves everyone.



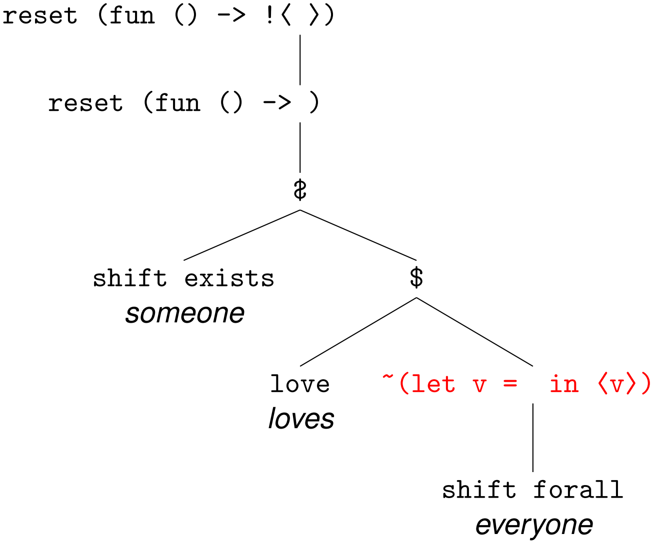
Inverse scope as quotation

“Someone loves everyone”.



Inverse scope as quotation

"Someone loves" everyone.



Outline

Past: Mixed quotation

Present: Quantifier scope

- Quotation for programming: code generation
- Control for programming: let insertion
- Control for linguistics: quantification
- Quotation for linguistics: inverse scope

- **Future: Rational metaprogramming**

Speaker and hearer model each other

Isn't it getting chilly in here?

Speaker and hearer model each other

Isn't it getting chilly in here?

A hotel cleaner enters a room and starts to clean it.
A female guest emerges from the shower.
The cleaner says "Excuse me sir" and exits.

Rational metaprogramming

To model the beliefs, desires, and intentions of agents who have
beliefs about each other's intentions,
about each other's desires about each other's beliefs,
and so on,

Rational metaprogramming

To model the beliefs, desires, and intentions of agents who have
beliefs about each other's intentions,
about each other's desires about each other's beliefs,
and so on,

we model

- intentions to perform actions as *programs*.
- beliefs as probability distributions.
  (weighted nondeterminism → stochastic programs)
- desires as utility functions.
  (rational choice → rational programs)

Rational metaprogramming

To model the beliefs, desires, and intentions of agents who have
beliefs about each other's intentions,
about each other's desires about each other's beliefs,
and so on,

we model

- intentions to perform actions as *programs*.
- beliefs as probability distributions.
  (weighted nondeterminism $\rightarrow$ stochastic programs)
- desires as utility functions.
  (rational choice $\rightarrow$ rational programs)

One agent's model of another is a probability distribution over (quoted) rational programs.

We need a modal type system and efficient self-interpretation.

Summary

Quotation goes well with effects (state, control, nondeterminism), **so that code does not have to be generated in lexical order.**

But we want a type system that prevents scope extrusion.

Multigrained theories of quotation:
the less intensional a theory,
the more cross-stage persistence it allows?

Levels of quotation are not quite levels of control operators.

Reckless let insertion

```
let test6 = ⟨fun x -> ~(reset (fun () ->
    power6 ⟨x⟩ 11))⟩
```

▶
```
val test6: ('a, int -> int) code
= ⟨fun x -> let r1 = x  * x  in
            let r2 = r1 * r1 in
            let r3 = r2 * r2 in (r3 * r1) * x⟩
```

Reckless let insertion

```
let test6 = ⟨fun x -> ~(reset (fun () ->
    power6 ⟨x⟩ 11))⟩
▶ val test6: ('a, int -> int) code
  = ⟨fun x -> let r1 = x  * x  in
              let r2 = r1 * r1 in
              let r3 = r2 * r2 in (r3 * r1) * x⟩

let test7a = ⟨fun x -> ~(reset (fun () ->
    ⟨let y = x + 1 in ~(power6 ⟨y⟩ 11)⟩)))⟩
▶ val test7a: ('a, int -> int) code
  = ⟨fun x -> let r1 = y  * y  in
              let r2 = r1 * r1 in
              let r3 = r2 * r2 in
              let y = x + 1 in (r3 * r1) * y⟩
```

Reckless let insertion

```
let test6 = ⟨fun x -> ~(reset (fun () ->
    power6 ⟨x⟩ 11))⟩
▶ val test6: ('a, int -> int) code
  = ⟨fun x -> let r1 = x  * x  in
              let r2 = r1 * r1 in
              let r3 = r2 * r2 in (r3 * r1) * x⟩

let test7b = ⟨fun x -> ~(reset (fun () ->
    ⟨let y = x + 1 in ~(reset (fun () -> power6 ⟨y⟩ 11))⟩))⟩
▶ val test7b: ('a, int -> int) code
  = ⟨fun x -> let y = x + 1 in
              let r1 = y  * y  in
              let r2 = r1 * r1 in
              let r3 = r2 * r2 in (r3 * r1) * y⟩
```