Lecture 24: Continuations, continued

CS 181O
Spring 2016
Kim Bruce

Continuation

- Meaning of linguistic context of expression called its *continuation*
- Expressions can have lots of continuations
  - Hence we'll make the continuation a parameter of the meaning.
  - We can think of providing an argument to a function
  - ... or a function to an argument!

What is a continuation?

- Continuation is provided to an expression so can get meaning of sentence.
- Continuation is function type returning type t
  - Continuation of NP is of type $e \to t$
  - Continuation of intransitive verb is $(e \to t) \to t$
  - Continuation of transitive verb is $(e \to e \to t) \to t$
- Meaning functions will now take continuations as an argument to get meaning.

Computations

- Computations are functions that take a continuation and give a meaning (of type r)
  - type Cont a r = a -> r
  - type Comp a r = Cont a r -> r
- Examples:
  - meaning of NP: $(e \to t) \to t$
  - meaning of IV, CN: $((e \to t) \to t) \to t$
  - meaning of TV: $((e \to e \to t) \to t) \to t$
  - meaning of ADJ: ?

Continuation-Passing Semantics

- Make all meaning take a continuation parameter k
  - Constant [[c]] ⇒ λk. k c
    - cpsConst:: a -> Comp a r
    - cpsConst c = \ k -> k c
  - *Trick to check work*: Get the original meaning by applying to identity function
    - (cpsConst c) (\x -> x) = (\ k -> k c) (\x -> x) = (\x -> x) c = c

Application?

- [[Dorothy cheered]]
  - [[Dorothy]] = λk:e → t. k dorothy:: Comp e t
    - where Comp e t = (e → t) → t
  - [[cheered]] = λk':(e → t) → t. k' cheered:: Comp (e → t) t
    - where Comp (e → t) t = ((e → t) → t) → t
  - [[Dorothy cheered]]: Comp t t
    - where Comp t t = (t → t) → t
    - [[Dorothy cheered]] = λk: t → t. ... ???

CpsApply

- cpsApply m n = λk . n (λb. m (λa. k (a b)))
  - result is a function that takes a continuation k.
  - To use k, must:
    - evaluate n with a continuation that takes the value b of n, and then
    - evaluates m with a continuation that takes the value a of m, and
    - finally applies k to the result of evaluating (a b)
- *Watch out, there is an alternative later!!*

CpsApply

- intSent_CPS (Sent np vp) =
  cpsApply (intVP_CPS vp) (intNP_CPS np)
  - Given continuation k:
  - Compute intNP_CPS np, call it b
  - Compute intVP_CPS vp, call it a
  - Apply k to (a b)
  - *Work out intSent_CPS(Sent Dorothy Cheered)*

Example

intSent_CPS(Sent Dorothy Cheered) =

cpsApply (intVP_CPS Cheered) (intNP_CPS Dorothy) =

cpsApply (λk':(e → t) → t. k' cheered) (λk:e → t. k dorothy) =

(λk''' . (λk:e → t. k dorothy) ((λb. (λk':(e → t) → t. k' cheered)

(λa. k''' (a b)))) =

(λk''' . (λk:e → t. k dorothy) ((λb.((λa. k''' (a b)) cheered)))) =

(λk''' . (λk:e → t. k dorothy) ((λb.(k''' (cheered b))))) =

(λk''' .k''' (cheered dorothy))

Example

intSent_CPS(Sent Dorothy Cheered) =

cpsApply (intVP_CPS Cheered) (intNP_CPS Dorothy) =

cpsApply (λk':. k' cheered) (λk. k dorothy) =

(λk''' . (λk. k dorothy) ((λb. (λk'. k' cheered) (λa. k''' (a b)))) =

(λk''' . (λk. k dorothy) ((λb.((λa. k''' (a b)) cheered)))) =

(λk''' . (λk. k dorothy) ((λb.(k''' (cheered b))))) =

(λk''' .k''' (cheered dorothy))

More CPS

- What about quantifiers?
  - [[everyone]] = λk. ∀x ((Person x) → k x)
  - [[someone]] = λk. ∃x ((Person x) ∧ k x)
  - *What is scope of x? Includes k!*
- Abstract to quantifiers:
  - [[every]] = λk λP. k(λQ.∀x ((Q x) → P x)
  - [[some]] = λk λP. k(λQ.∃x (Q x) ∧ P x)
  - [[the]] = λk λP. k(λQ.∃x ((...Q x) ∧ P x)
  - [[no]] = λk λP. k(λQ.¬∃x ((Q x) ∧ P x)

Example

[[every person]]

= (λk λP. k(λQ.∀x ((Q x) → P x))))(λk'. k' Person)

= (λP. (λk'. k' Person)(λQ.∀x ((Q x) → P x))

= (λP.(λQ.(∀x (Q x) → P x)Person)

= (λP.((∀x (Person x) → P x)

has type (e → t) → t

Same value as everyone, as expected!

Transitive Verbs

intTV_CPS Helped = cpsConst help

$= \lambda k.k\ \text{helped}$

where helped: $e \rightarrow e \rightarrow t$

First argument is object, second is subject!

Scope Reversal

- Can use different apply function:
  - cpsApply' :: Comp (a -> b) r -> Comp a r -> Comp b r
  - cpsApply' m n = $\lambda k.\ m\ (\lambda a.\ n\ (\lambda b.\ k\ (a\ b)))$
- Compared to original:
  - cpsApply m n = $\lambda k\ .\ n\ (\lambda b.\ m\ (\lambda a.\ k\ (a\ b)))$
- What does it mean in practice
  - Everyone helped someone.

From the Text:

Using cpsApply:

$[[\text{everyone}]] = \lambda k'.\forall x.\ ((\text{Person } x) \rightarrow (k'\ x))$
$[[\text{someone}]] = \lambda k'.\ \exists x.\ ((\text{Person } x) \wedge (k'\ x))$

$[[\text{helped someone}]] = \text{cpsApply } ([[\text{helped}]])(\text{someone}]])$
$= \lambda k.([[\text{someone}]](\lambda n.\ [[\text{helped}]](\lambda m.k\ (m\ n))))$
$= \lambda k.((\lambda k'.\ \exists x.\ ((\text{Person } x) \wedge (k'\ x))(\lambda n.\ [[\text{helped}]](\lambda m.k\ (m\ n))))$
$= \lambda k.(\exists x.\ ((\text{Person } x) \wedge (\lambda n.\ [[\text{helped}]](\lambda m.k\ (m\ n)))x))$
$= \lambda k.(\exists x.\ ((\text{Person } x) \wedge ([[\text{helped}]](\lambda m.k\ (m\ x))))$
$= \lambda k.(\exists x.\ ((\text{Person } x) \wedge ((\lambda k'.\ k'\ \text{help})(\lambda m.k\ (m\ x))))$
$= \lambda k.(\exists x.\ ((\text{Person } x) \wedge (\lambda m.k\ (m\ x))\text{help}))$
$= \lambda k.(\exists x.\ (\text{Person } x) \wedge (k\ (\text{help } x)))$

From the Text:

$[[\text{everyone}]] = \lambda k'.\forall x.\ ((\text{Person } x) \rightarrow (k'\ x))$
$[[\text{someone}]] = \lambda k'.\ \exists x.\ ((\text{Person } x) \wedge (k'\ x))$

$[[\text{helped someone}]] = \lambda k'.(\exists y.\ (\text{Person } y) \wedge (k'\ (\text{help } y)))$

$[[\text{everyone helped someone}]] = \text{cpsApply}([[\text{helped someone}]])([[\text{everyone}]])$
$= \lambda k.([[\text{everyone}]](\lambda b.\ [[\text{helped someone}]](\lambda a.k\ (a\ b))))$
$= \lambda k.(\lambda k'.\forall x.\ ((\text{Person } x) \rightarrow k'\ x)(\lambda b.\ [[\text{helped someone}]](\lambda a.k\ (a\ b))))$
$= \lambda k.(\forall x.\ ((\text{Person } x) \rightarrow (\lambda b.\ [[\text{helped someone}]](\lambda a.k\ (a\ b))x))$
$= \lambda k.(\forall x.\ ((\text{Person } x) \rightarrow ([[\text{helped someone}]](\lambda a.k\ (a\ x))))$
$= \lambda k.(\forall x.\ ((\text{Person } x) \rightarrow (\lambda k'.(\exists y.\ (\text{Person } y) \wedge (k'\ (\text{help } y))))(\lambda a.k\ (a\ x))))$
$= \lambda k.(\forall x.\ ((\text{Person } x) \rightarrow (\exists y.\ (\text{Person } y) \wedge ((\lambda a.k\ (a\ x))\ (\text{help } y)))))$
$= \lambda k.(\forall x.\ ((\text{Person } x) \rightarrow (\exists y.\ (\text{Person } y) \wedge (k\ ((\text{help } y)\ x)))))$

From the Text:

Using cpsApply':

$[[\text{everyone}]] = \lambda k'.\forall x.\ ((\text{Person } x) \rightarrow (k'\ x))$
$[[\text{someone}]] = \lambda k'.\ \exists x.\ ((\text{Person } x) \wedge (k'\ x))$

$[[\text{helped someone}]] = \text{cpsApply}'\ ([[\text{helped}]])([[\text{someone}]])$
$= \lambda k.([[\text{helped}]](\lambda a.[[\text{someone}]](\lambda b.k\ (a\ b))))$
$= \lambda k.(\lambda k'.\ k'\ \text{help})(\lambda a.[[\text{someone}]](\lambda b.k\ (a\ b))))$
$= \lambda k.(\lambda a.[[\text{someone}]](\lambda b.k\ (a\ b)))\text{help})$
$= \lambda k.([[\text{someone}]](\lambda b.k\ (\text{help } b)))$
$= \lambda k.(\lambda k'.\ \exists x.\ ((\text{Person } x) \wedge (k'\ x))(\lambda b.k\ (\text{help } b)))$
$= \lambda k.(\exists x.\ ((\text{Person } x) \wedge ((\lambda b.k\ (\text{help } b))\ x)))$
$= \lambda k.(\exists x.\ (\text{Person } x) \wedge (k\ (\text{help } x)))$

Exactly as before!

From the Text:

$[[\text{everyone}]] = \lambda k'.\forall x.\ ((\text{Person } x) \rightarrow (k'\ x))$
$[[\text{someone}]] = \lambda k'.\ \exists x.\ ((\text{Person } x) \wedge (k'\ x))$

$[[\text{helped someone}]] = \lambda k'.(\exists y.\ (\text{Person } y) \wedge (k'\ (\text{help } y)))$

$[[\text{everyone helped someone}]] = \text{cpsApply}'([[\text{helped someone}]])([[\text{everyone}]])$
$= \lambda k.([[\text{helped someone}]](\lambda a.[[\text{everyone}]](\lambda b.k\ (a\ b))))$
$= \lambda k.\underline{(\lambda k'.(\exists y.\ (\text{Person } y) \wedge (k'\ (\text{help } y))))}(\lambda a.\ [[\text{everyone}]](\lambda b.k\ (a\ b))))$
$= \lambda k.(\exists y.\ (\text{Person } y) \wedge \underline{(\lambda a.\ [[\text{everyone}]](\lambda b.k\ (a\ b))}(\text{help } y))$
$= \lambda k.(\exists y.\ (\text{Person } y) \wedge ([[\text{everyone}]](\lambda b.k\ (\underline{(\text{help } y)}\ b))))$
$= \lambda k.(\exists y.\ (\text{Person } y) \wedge\ \underline{(\lambda k'.\forall x.\ ((\text{Person } x) \rightarrow (k'\ x)}) (\lambda b.k\ ((\text{help } y)\ b))))$
$= \lambda k.(\exists y.\ (\text{Person } y) \wedge (\forall x.\ ((\text{Person } x) \rightarrow \underline{((\lambda b.k\ ((\text{help } y)\ b))}x)))$
$= \lambda k.(\exists y.\ (\text{Person } y) \wedge (\forall x.\ ((\text{Person } x) \rightarrow (k\ ((\text{help } y)\ \underline{x})))))$

Bottom Line

- cpsApply expands subject first, with object expanded inside.
- cpsApply' does opposite
- Allows us to capture both expressions of quantifiers.

More continuations

- Can be helpful in handling coordination
- Already know how to make sense of sentential operators: and, or, not
  - Interpreted in predicate logic with $\wedge$, $\vee$, $\neg$
- But they also appear as operators on other grammatical features

Coordination

- NP: John and Mary went to the store
  - John went to the store and Mary went to the store
- V: Mary danced and sang all night
  - Mary danced all night and Mary sang all night
- Adj: The ball was big and red
- VP: John kicked the ball and ran down the field
  - John kicked the ball and John ran down the field
- Ann baked and Betty ate all the cookies.

Meaning via Continuations

- What is context around conjunctive phrase?
  - Mary danced and sang all night
  - k = λx. Mary x all night
  - k (danced and sang) = k(danced) and k(sang)

  - intCON_CPS And = $\lambda k\ \lambda m\ \lambda n.\ k(m) \wedge k(n)$
  - intCON_CPS Or = $\lambda k\ \lambda m\ \lambda n.\ k(m) \vee k(n)$

Still issues

- Chris and Betty met at the fair
  - Chris met at the fair $\wedge$ Betty met at the fair????
- Different meaning of “and”
  - Individuals or group

Questions?