Continuation semantics i[1]

Patrick D. Elliott & Martin Hackl

February 3, 2020

Roadmap

Some notation conventions

Generally speaking, I'll be assuming Heim & Kratzer 1998 as background, but I'll depart from their notation slightly.

Expressions in the meta-language will be typeset in sans serif.

$$[\![[_{\text{DP}}\ \text{John}]]\!] \coloneqq \underbrace{\mathsf{John}}_{\text{individual}}$$

Seeing as its primitive, we'll treat white-space as function application, e.g.:

(1) $(\lambda x\ .\ \mathsf{left}\ x)\ \mathsf{paul} = \mathsf{left}\ \mathsf{paul}$

Function application associates to the *left*:

(2) $(\lambda x\ .\ \lambda y\ .\ y\ \mathsf{likes}\ x)\ \mathsf{paul}\ \mathsf{sophie} \equiv ((\lambda x\ .\ \lambda y\ .\ y\ \mathsf{likes}\ x)\ \mathsf{paul})\ \mathsf{sophie}$

We'll write *types* in a `fixed width font`. We have our familiar primitive types...

(3) $\mathtt{type} \coloneqq \mathtt{e} \mid \mathtt{t} \mid \mathtt{s} \mid \ldots$

...and of course *function types*. Unlike Heim & Kratzer (1998), who use $\langle . \rangle$ as the constructor for a function type, we'll be using the (more standard (outside of linguistics!)) arrow constructor ($\rightarrow$):

(4) $\langle \mathtt{e}, \mathtt{t} \rangle \equiv \mathtt{e} \rightarrow \mathtt{t}$

The constructor for function types *associates to the right*:

(5) $\mathtt{e} \rightarrow \mathtt{e} \rightarrow \mathtt{t} \equiv \mathtt{e} \rightarrow (\mathtt{e} \rightarrow \mathtt{t})$

[1] 24.979: Topics in semantics

Getting high:

scope, projection, and evaluation order

The Partee triangle

(6) $\mathsf{LIFT}x \coloneqq \lambda k \ . \ k \ x$ $\qquad\qquad \mathsf{LIFT} : \mathtt{e} \to (\mathtt{e} \to \mathtt{t}) \to \mathtt{t}$

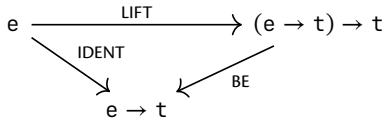
(7) $\mathsf{IDENT} \ x \coloneqq \lambda y \ . \ y = x$ $\qquad\qquad \mathsf{IDENT} : \mathtt{e} \to \mathtt{e} \to \mathtt{t}$

(8) $\mathsf{BE} \ Q \coloneqq \lambda x \ . \ Q \ (\lambda y \ . \ y = x)$ $\qquad\qquad \mathsf{BE} : ((\mathtt{e} \to \mathtt{t}) \to \mathtt{t}) \to \mathtt{e} \to \mathtt{t}$

Commutative diagrams

The *Partee triangle* is a *commutative diagram*. We say that a diagram *commutes* if, when there are multiple paths between two points, those paths are equivalent. The equivalence in (10) is therefore expressed by the triangle.

(9) The Partee triangle[2]



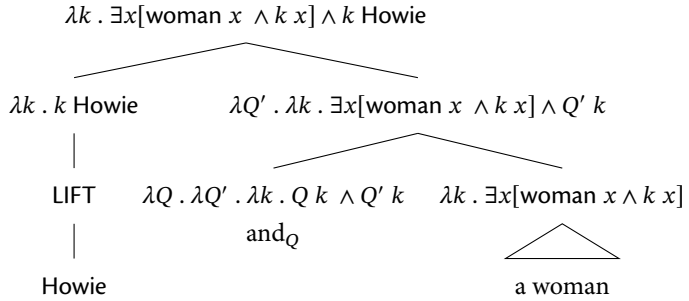
(10) $\mathsf{ident} \equiv \mathsf{BE} \circ (\uparrow)$

Observation: expressions quantificational and non-quantificational DPs can be coordinated:

(11) [Howie and a woman] entered the club

LIFT allows something that, by virtue of its quantificational nature, is an *inherent* scope-taker, to combine with something that *isn't*:[3]

(12)



[2] Partee 1986

[3] If you're familiar with Partee & Rooth 2012 you'll notice that the *and* that coordinates quantificational DPs (written here as and_Q), is just the result of applying their *generalized conjunction* rule. We'll return to generalized conjunction, and the connection to coninuations in §.

Generalizing the triangle

Based on the way in which LIFT and friends are defined, in theory we could replace e with *any* type. Let's give a more general statement of LIFT and friends as polymorphic functions:

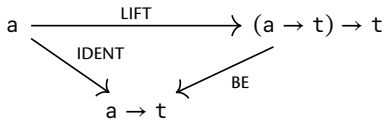
(13) $\mathsf{LIFT}x \coloneqq \lambda k\ .\ k\ x$ $\qquad\qquad \mathsf{LIFT} : \mathtt{a} \to (\mathtt{a} \to \mathtt{t}) \to \mathtt{t}$

(14) $\mathsf{IDENT}\ x \coloneqq \lambda y\ .\ y = x$ $\qquad\qquad \mathsf{IDENT} : \mathtt{a} \to \mathtt{a} \to \mathtt{t}$

(15) $\mathsf{BE}\ Q \coloneqq \lambda x\ .\ Q\ (\lambda y\ .\ y = x)$ $\qquad\qquad \mathsf{BE} : ((\mathtt{a} \to \mathtt{t}) \to \mathtt{t}) \to \mathtt{a} \to \mathtt{t}$

The diagram, of course, still commutes:

(16) The (generalized) Partee triangle

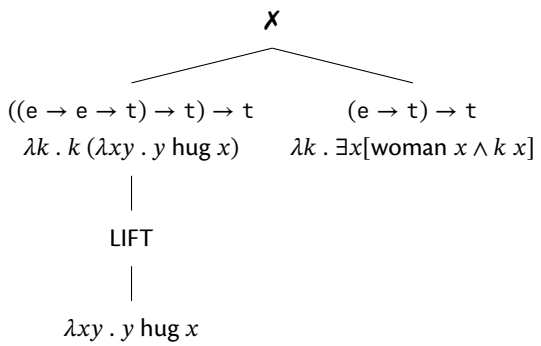


Why might a polymorphic LIFT be useful? Recall that we used a typed instantiation of LIFT in order to allow a quantificational thing to combine with a non-quantificational thing. Polymorphic LIFT allows us to type-lift, e.g., a function that takes multiple arguments:

(17) $\mathsf{LIFT}\ (\lambda xy\ .\ y\ \mathsf{hug}\ x) = \overbrace{\lambda k\ .\ k\ (\lambda xy\ .\ y\ \mathsf{hug}\ x)}^{((\mathtt{e} \to \mathtt{e} \to \mathtt{t}) \to \mathtt{t}) \to \mathtt{t}}$

One tantalizing possibility is that this allow us to combine a non-quantificational transitive verb with a quantificational DP.

(18)



Unfortunately, assuming the usual inventory of composition rules (i.e., *function*

application, *predicate modification*, and *predicate abstraction*), we're stuck.[4] So, **let's invent a new one**.

Here's the intuition we're going to pursue. Let's look again at the types. One way of thinking about what LIFT as follows: it takes an a-type thing and adds a "wrapper". Quantificational DPs, on the other hand come "pre-wrapped".

- LIFT ⟦hug⟧ : ($\boxed{(e \to e \to t)}$ → t) → t
- ⟦a woman⟧ : ($\boxed{e}$ → t) → t

If we look at the wrapped-up types, we see a *function* from individuals, and an individual – namely, two things that can combine via function application.

What we need to accomplish is the following:

- Unwrap lifted *hug*.
- Unwrap the *a woman*.
- Use function application to combine the unwrapped values.
- Finally, wrap the result back up! Think of the quantificational meaning as being like a taco – it isn't really a taco without the wrapper, therefore we don't want to throw the wrapper away.

In order to accomplish this, we'll define a new composition rule: Scopal Function Application (SFA). We're going to define SFA in terms of Function Application (FA); we haven't been explicit about how FA is defined yet, so let's do that now. We'll write FA as the infix operator A.[5]

(19) Function Application (FA) (def.)

a. f A x := f x — A : (a → b) → a → b

b. x A f := f x — A : a → (a → b) → b

Here, function application is made bidirectional by *overloading* – we've defined forwards and backwards application, and given them the same function name.

[4] Other, more exotic composition rules such as *restrict* won't help either. Take my word for this!

[5]

different ways of making function application bidirectional

Scopal Function Application (SFA)

We'll write SFA as the infix operator S. Note that, since A is overloaded already, and S is defined in terms of A, S gets overloaded too.

(20) Scopal Function Application (SFA) (def.)

$m\ \mathsf{S}\ n \coloneqq \lambda k\ .\ m\ (\lambda a\ .\ n\ (\lambda b\ .\ k\ (a\ \mathsf{A}\ b)))$

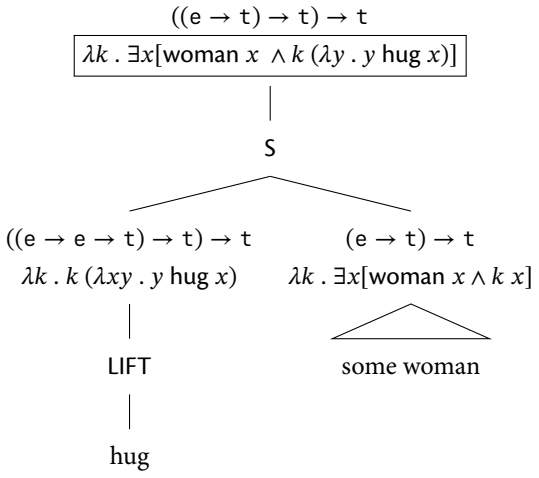
$\mathsf{S} : (((\mathtt{a} \to \mathtt{b}) \to \mathtt{t}) \to \mathtt{t}) \to ((\mathtt{a} \to \mathtt{t}) \to \mathtt{t}) \to (\mathtt{b} \to \mathtt{t}) \to \mathtt{t}$

$\mathsf{S} : ((\mathtt{a} \to \mathtt{t}) \to \mathtt{t}) \to (((\mathtt{a} \to \mathtt{b}) \to \mathtt{t}) \to \mathtt{t}) \to (\mathtt{b} \to \mathtt{t}) \to \mathtt{t}$

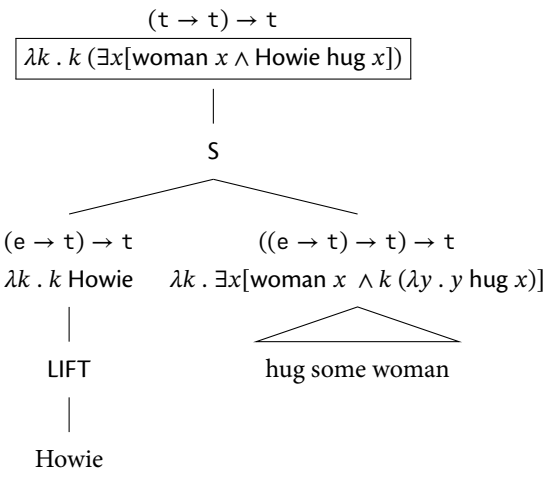
Now, let's illustrate how SFA plus generalized LIFT allows to compose a quantificational thing with non-quantificational things, using a simple example sentence:

(21) Howie hugged some woman.

(22) Step 1: compose *some woman* with LIFT-ed *hug*.



(23) Step 2: compose the resulting VP-denotation with LIFT-ed *Howie*



We're now tantalizingly close to deriving the right kind of object for the sentential meaning (namely, something of type t). Only, what we have is something of type (⌈t⌉ → t) → t, i.e., a truth value in a "wrapper". How do we get back the wrapped up value? We saturate the k argument with the identity function. We'll call this operation LOWER.[6]

(24) LOWER (def.)
LOWER m := m id LOWER : ((a → a) → a) → a

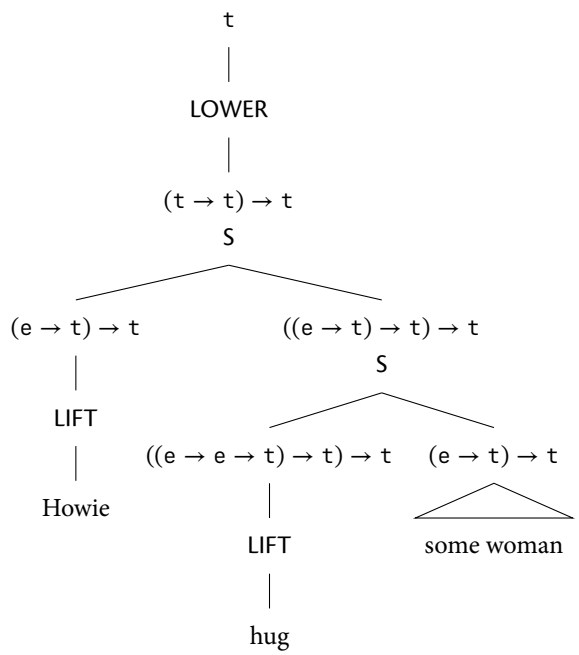
Applying LOWER to the final value in (23) gives us a type t sentential meaning.

(25) LOWER (λk . k ($\exists x$[Howie hug x]))
= (λk . k($\exists x$[Howie hug x])) id
= id ($\exists x$[Howie hug x])
= $\exists x$[Howie hug x]

Let's zoom back out and see how all the pieces fit together by looking at the *graph of the derivation*.

[6] Here we've given LOWER the maximally polymorphic type compatible with the function definition; in fact, all we need is (LOWER : ((t → t) → t) → t).

(26)



So far, we've provided an account of how quantificational things compose with non-quantificational things, by making use of...

- ...an independently motivated type-shifting rule (LIFT)...
- ...a way to apply LIFT-ed values (S)...
- ...and a way to get an ordinary value back from a LIFT-ed value (LOWER).

This seems pretty nice, but as I'm sure you've noticed, things are quickly going to get pretty cumbersome with more complicated sentences, especially with multiple quantifiers. Before we go any further, let's introduce some notational conveniences.

Towers

We've been using the metaphor of a *wrapper* for thinking about what LIFT does to an ordinary semantic value. Let's make this a bit more transparent by introducing a new *type constructor* for LIFT-ed values.[7]

(27) $\mathsf{K_t\ a := (a \to t) \to t}$

[7] A *type constructor* is just a function from a type to a new type – here, it's a rule for taking any type a and returning the type of the corresponding LIFT-ed value.

- Quantificational DPs are therefore of type $\mathsf{K_t\ e}$ (inherently).
- LIFT takes something of type a, and lifts it into something of type $\mathsf{K_t\ a}$.

Rather than dealing with *flat* expressions of the simply-typed lambda calculus, which will become increasingly difficult to reason about, we'll follow Barker & Shan 2014 in using *tower notation*.[8]

Let's look again at the meaning of a quantificational DP. The k argument which acts as the *wrapper* is called the *continuation argument*.

(28) $[\![\text{some woman}]\!] \coloneqq \lambda k\ .\ \exists x[\mathsf{woman}\ x\ \wedge k\ x]$

(29) $[\![\text{some woman}]\!] \coloneqq \dfrac{\exists x[\mathsf{woman}\ x \wedge []]}{x}$

(30) $\mathsf{LIFT}\ ([\![\text{hug}]\!]) = \dfrac{[]}{\lambda xy\ .\ y\ \mathsf{hug}\ x}$

In general:

(31) $\dfrac{f\ []}{x} \coloneqq \lambda k\ .\ f\ (k\ x)$

We can use tower notation for types too:

(32) $\dfrac{\mathsf{b}}{\mathsf{a}} \coloneqq (\mathsf{a} \to \mathsf{b}) \to \mathsf{b}$

We can now redefine our type constructor $\mathsf{K_t}$, and our type-shifting operations using our new, much more concise, tower notation. These will be our canonical definitions from now on. We'll also start abbreviating a LIFT-ed value a as $a^{\uparrow}$ and a LOWER-ed value b as $b^{\downarrow}$.

(33) The continuation type constructor $\mathsf{K_t}$ (def.)

$$\mathsf{K_t\ a} \coloneqq \frac{\mathsf{t}}{\mathsf{a}}$$

[8] To my mind, one of Barker & Shan's central achievements is simply the introduction of an accessible notational convention for reasoning about the kinds of lifted meanings we're using here.

(34) LIFT (def.)[9]

$$a^{\uparrow} := \frac{[]}{a} \qquad (\uparrow) : \mathsf{a} \to \mathsf{K_t\ a}$$

(35) Scopal Function Application (SFA) (def.)[10]

$$\frac{f\ []}{x}\ \mathsf{S}\ \frac{g\ []}{y} := \frac{f\ (g\ [])}{x\ \mathsf{A}\ y} \qquad \mathsf{S} : \mathsf{K_t\ (a \to b) \to K_t\ a \to K_t\ b}$$

(36) LOWER (def.)[11]

$$\left(\frac{f\ []}{p}\right)^{\downarrow} \qquad (\downarrow) : \mathsf{K_t\ t \to t}$$

In order to see the tower notation in action, let's go through an example involving multiple quantifiers, and show how continuation semantics derives the surface scope reading:

(37) Some knight chased every thief.

First, we combine *every thief* with lifted *chase* via SFA:

(38)

$$\boxed{\frac{\forall x[\text{thief } x \to []]}{\lambda y\ .\ y \text{ chase } x}}$$

S

$$\frac{[]}{\lambda xy.y \text{ chase } x} \qquad \frac{\forall x[\text{thief } x \to []]}{x}$$

Next, the (boxed) VP value combines with *some knight* via SFA:

(39)

$$\boxed{\frac{\exists y[\text{knight } y \wedge \forall x[\text{thief } x \to []]]}{y \text{ chase } x}}$$

S

$$\frac{\exists y[\text{knight } y \wedge []]}{y} \qquad \frac{\forall x[\text{thief } x \to []]}{\lambda y\ .\ y \text{ chase } x}$$

chase every thief

Finally, the resulting tower is collapsed via *lower*:

(40)

$$\boxed{\exists y[\text{knight } y \wedge \forall x[\text{thief } x \to y \text{ chase } x]]}$$

$$\left(\frac{\exists y[\text{knight } y \wedge \forall x[\text{thief } x \to []]]}{y \text{ chase } x}\right)^{\downarrow}$$

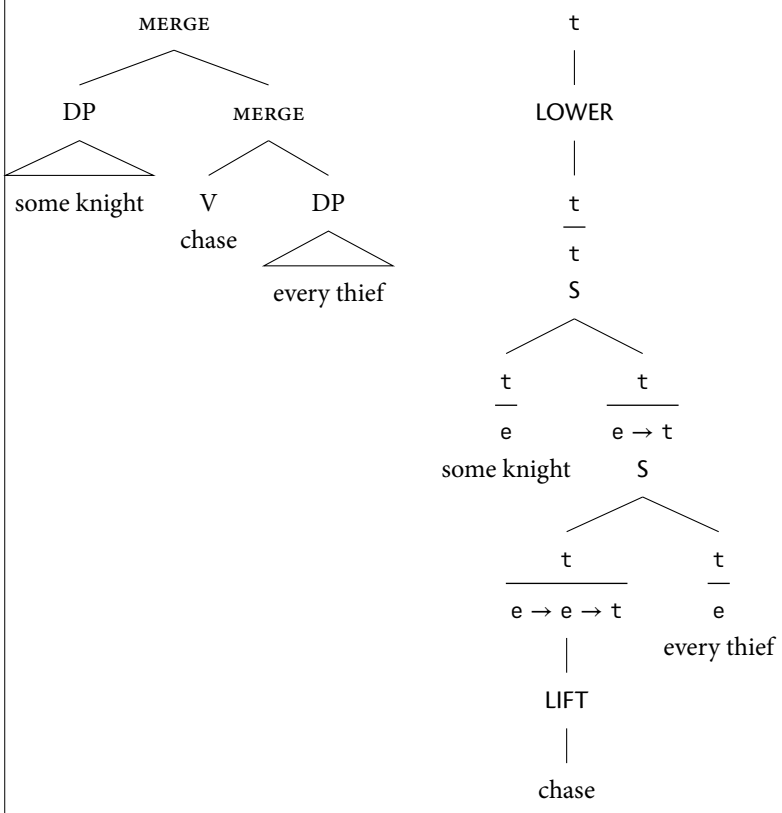
Let's zoom out and look at the *graph of the syntactic derivation* alongside the

[9] Thinking in terms of towers, LIFT takes a value a and returns a "trivial" tower, i.e., a tower with an empty top-story.

[10] SFA takes two scopal values – one with a function on the bottom floor, and the other with an argument on the bottom floor – and combines them by (i) doing function application on the bottom floor, and (ii) *sequencing* the scope-takers.

[11] LOWER *collapses the tower*, applying whatever is on the top story to whatever is on the bottom story.

graph of the semantic derivation.



Continuations beyond DPs

Generalized (con/dis)junction

generalized conjunction as continuised conjunction.

References

Barker, Chris & Chung-chieh Shan. 2014. *Continuations and natural language* (Oxford studies in theoretical linguistics 53). Oxford University Press. 228 pp.

Charlow, Simon. 2014. *On the semantics of exceptional scope*.

Charlow, Simon. 2018. A modular theory of pronouns and binding. unpublished manuscript.

Heim, Irene & Angelika Kratzer. 1998. *Semantics in generative grammar* (Blackwell textbooks in linguistics 13). Malden, MA: Blackwell. 324 pp.
Kiselyov, Oleg. 2017. Applicative abstract categorial grammars in full swing. In Mihoko Otake et al. (eds.), *New frontiers in artificial intelligence* (Lecture Notes in Computer Science), 66–78. Springer International Publishing.
Mcbride, Conor & Ross Paterson. 2008. Applicative programming with effects. *Journal of Functional Programming* 18(1).
Partee, Barbara. 1986. Noun-phrase interpretation and type-shifting principles. In J. Groenendijk, D. de Jongh & M. Stokhof (eds.), *Studies in discourse representation theory and the theory of generalized quantifiers*, 115–143. Dordrecht: Foris.
Partee, Barbara & Mats Rooth. 2012. Generalized conjunction and type ambiguity. In, Reprint 2012, 361–383. Berlin, Boston: De Gruyter.

Continuations from a categorical perspective

The triple $(\mathsf{K_t}, \uparrow, \mathsf{S})$ is an *applicative functor*, a highly influential notion in the literature on functional programming (Mcbride & Paterson 2008); for applications in linguistic semantics see Kiselyov 2017, Charlow 2018.

Cite my applicatives paper here

Technically speaking, an applicative functor is a type constructor (here $\mathsf{K_t}$), together with two functions $(\uparrow) : \mathsf{a} \to \mathsf{K_t\ a})$ and $(\mathsf{S} : \mathsf{K_t}\ (\mathsf{a} \to \mathsf{b}) \to \mathsf{K_t\ a} \to \mathsf{K_t\ b})$ obeying the following laws:

Check the applicative laws

(41) **Homomorphism**
$f^{\uparrow}\ \mathsf{S}\ x^{\uparrow} \equiv (f\ x)^{\uparrow}$

(42) **Interchange**
$(\lambda k\ .\ k\ x)^{\uparrow}\ \mathsf{S}\ m \equiv m \mathsf{S}\ x^{\uparrow}$

(43) **Identity**
$id^{\uparrow}\ \mathsf{S}\ m \equiv m$

(44) **Composition**
$(\circ)^{\uparrow}\ \mathsf{S}\ u\ \mathsf{S}\ v\ \mathsf{S}\ w \equiv u\ \mathsf{S}\ (v\ \mathsf{S}\ w)$

You can verify for yourselves that the triple $(\mathsf{K_t}, \uparrow, \mathsf{S})$ obeys the applicative laws – we'll call it the *continuation applicative*.

A related, more powerful abstraction from the functional programming literature is *monads*. There is a growing body of literature in linguistic semantics that explicitly makes use of monads (Charlow 2014,). A monad, like an applicative functor, is defined as a triple consisting of a type constructor and

cite others

two functions. Monads are strictly speaking more powerful than applicative functors; that is to say, if you have a monad you are guaranteed to have an applicative functor, but not vice versa.

In, e.g., Barker & Shan (2014), and much of the existing literature in linguistic semantics making use of continuations, they are presented in their applicative guise; in the functional programming literature however (and especially in `haskell`) continuations are more widely used in their monadic guise. A monad is a triple $(\mathtt{K_t}, (\uparrow), \mu)$ – the crucial addition here is *join* (μ).

(45) *join* (def.)

a. $\mu : \mathtt{K_t}\ (\mathtt{K_t}\ \mathtt{a}) \rightarrow \mathtt{K_t}\ \mathtt{a}$

b. $\mu\ m = \lambda k\ .\ m\ (\lambda K\ .\ K\ k)$

(46) $\mu\ (\lambda k\ .\exists x[k\ (\lambda l\ .\ \forall y[l\ (x\ \textsf{likes}\ y)])]) = \lambda j\ .\ \exists x[\forall y[j\ (x\ \textsf{likes}\ y)])]$

LaTeX dojo

Here is the macro I use to typeset towers in LaTeX. Declare this in your preamble. You'll need the `booktabs` and `xparse` packages too.

```
\NewDocumentCommand\semtower{mm}{
  \begin{tabular}[c]{@{\,}c@{\,}}
    \(#1\)
    \\
    \midrule
    \(#2\)
    \\
  \end{tabular}
}
```

A simple two-level tower can now be typeset as follows:

```
$$\semtower{f []}{x}$$
```

Resulting in:

$$\begin{array}{c} f\ [] \\ \hline x \end{array}$$