10—Modal Logic IV; Lambda Calculus

UIT2206: The Importance of Being Formal

Martin Henz

March 27, 2013

Generated on Wednesday 27th March, 2013, 09:57

Syntax of Basic Modal Logic

$$
\begin{aligned}
\phi \quad ::= \quad & \top \mid \bot \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \\
& \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi) \\
& \mid (\phi \leftrightarrow \phi) \\
& \mid (\Box\phi) \mid (\Diamond\phi)
\end{aligned}
$$

Kripke Models

Definition

A model $\mathcal{M}$ of propositional modal logic over a set of propositional atoms A is specified by three things:

1. A W of *worlds*;
2. a relation R on W, meaning $R \subseteq W \times W$, called the *accessibility relation*;
3. a function $L : W \rightarrow A \rightarrow \{T, F\}$, called *labeling function*.

When is a formula true in a possible world?

Definition

Let $\mathcal{M} = (W, R, L)$, $x \in W$, and ϕ a formula in basic modal logic. We define $x \Vdash \phi$ via structural induction:

- $x \Vdash \top$
- $x \nVdash \bot$
- $x \Vdash p$ iff $p \in L(x)(p) = T$
- $x \Vdash \neg\phi$ iff $x \nVdash \phi$
- $x \Vdash \phi \wedge \psi$ iff $x \Vdash \phi$ and $x \Vdash \psi$
- $x \Vdash \phi \vee \psi$ iff $x \Vdash \phi$ or $x \Vdash \psi$
- ...

When is a formula true in a possible world?

Definition (continued)

Let $\mathcal{M} = (W, R, L)$, $x \in W$, and ϕ a formula in basic modal logic. We define $x \Vdash \phi$ via structural induction:

- ...
- $x \Vdash \phi \rightarrow \psi$ iff $x \Vdash \psi$, whenever $x \Vdash \phi$
- $x \Vdash \phi \leftrightarrow \psi$ iff ($x \Vdash \phi$ iff $x \Vdash \psi$)
- $x \Vdash \Box\phi$ iff for each $y \in W$ with $R(x, y)$, we have $y \Vdash \phi$
- $x \Vdash \Diamond\phi$ iff there is a $y \in W$ such that $R(x, y)$ and $y \Vdash \phi$.

A Range of Modalities

In a particular context $\Box\phi$ could mean:

- It is necessarily true that ϕ
- It ought to be that ϕ
- Agent *Q* believes that ϕ
- Agent *Q* knows that ϕ

Since $\Diamond\phi \equiv \neg\Box\neg\phi$, we can infer the meaning of $\Diamond$ in each context.

A Range of Modalities

From the meaning of $\Box\phi$, we can conclude the meaning of $\Diamond\phi$, since $\Diamond\phi \equiv \neg\Box\neg\phi$:

$\Box\phi$	$\Diamond\phi$
It is necessarily true that ϕ	It is possibly true that ϕ
It ought to be that ϕ	It is permitted to be that ϕ
Agent *Q* believes that ϕ	ϕ is consistent with *Q*'s beliefs
Agent *Q* knows that ϕ	For all *Q* knows, ϕ

Reflexivity and Transitivity

Theorem

The following statements are equivalent:

- R is reflexive;
- $\mathcal{F}$ satisfies $\Box\phi \to \phi$;
- $\mathcal{F}$ satisfies $\Box p \to p$;

Theorem

The following statements are equivalent:

- R is transitive;
- $\mathcal{F}$ satisfies $\Box\phi \to \Box\Box\phi$;
- $\mathcal{F}$ satisfies $\Box p \to \Box\Box p$;

Formula Schemes and Properties of *R*

name	formula scheme	property of *R*
T	$\Box\phi \to \phi$	reflexive
B	$\phi \to \Box\Diamond\phi$	symmetric
D	$\Box\phi \to \Diamond\phi$	serial
4	$\Box\phi \to \Box\Box\phi$	transitive
5	$\Diamond\phi \to \Box\Diamond\phi$	Euclidean
	$\Box\phi \leftrightarrow \Diamond\phi$	functional
	$\Box(\phi \wedge \Box\phi \to \psi) \vee \Box(\psi \wedge \Box\psi \to \phi)$	linear

Which Formula Schemes to Choose?

Definition

Let $\mathcal{L}$ be a set of formula schemes and $\Gamma \cup \{\psi\}$ a set of formulas of basic modal logic.

- A set of formula schemes is said to be *closed* iff it contains all substitution instances of its elements.
- Let $\mathcal{L}_c$ be the smallest closed superset of $\mathcal{L}$.
- Γ entails ψ in $\mathcal{L}$ iff $\Gamma \cup \mathcal{L}_c$ semantically entails ψ. We say $\Gamma \models_{\mathcal{L}} \psi$.

Examples of Modal Logics: K

K is the weakest modal logic, $\mathcal{L} = \emptyset$.

Examples of Modal Logics: KT45

$\mathcal{L} = \{T, 4, 5\}$

Used for reasoning about knowledge.

name	formula scheme	property of R
T	$\Box\phi \to \phi$	reflexive
4	$\Box\phi \to \Box\Box\phi$	transitive
5	$\Diamond\phi \to \Box\Diamond\phi$	Euclidean

- T: Truth: agent Q only knows true things.
- 4: Positive introspection: If Q knows something, he knows that he knows it.
- 5: Negative introspection: If Q doesn't know something, he knows that he doesn't know it.

Explanation of Negative Introspection

name	formula scheme	property of *R*
...	...	...
5	$\Diamond\phi \rightarrow \Box\Diamond\phi$	Euclidean

$$\begin{array}{rcl} \Diamond\phi & \rightarrow & \Box\Diamond\phi \\ \Diamond\neg\psi & \rightarrow & \Box\Diamond\neg\psi \\ \neg\Box\neg\neg\psi & \rightarrow & \Box\neg\Box\neg\neg\psi \\ \neg\Box\psi & \rightarrow & \Box\neg\Box\psi \end{array}$$

If *Q* doesn't know ψ, he knows that he doesn't know ψ.

Correspondence for KT45

Accessibility relations for KT45

KT45 hold if and only if *R* is reflexive (T), transitive (4) and Euclidean (5).

Fact on such relations

A relation is reflexive, transitive and Euclidean iff it is reflexive, transitive and symmetric, i.e. iff it is an equivalence relation.

Examples of Modal Logics: KD45

$\mathcal{L} = \{D, 4, 5\}$

name	formula scheme	property of R
D	$\Box\phi \to \Diamond\phi$	serial
4	$\Box\phi \to \Box\Box\phi$	transitive
5	$\Diamond\phi \to \Box\Diamond\phi$	Euclidean

- D: agent Q only believes believable things.
- 4: positive introspection: If Q believes something, he believes that he believes it.
- 5: Negative introspection: If Q doesn't believe something, he believes that he doesn't believe it.

Correspondence for KD45

Accessibility relations for KT4

KT4 hold if and only if *R* is serial (D), transitive (4), and Euclidean (5).

Dashed Boxes

Idea

In addition to proof boxes for assumptions, we introduce *blue boxes* that express knowledge about an *arbitrary accessible world*.

Rules about blue boxes

- Whenever $\Box\phi$ occurs in a proof, ϕ may be put into a subsequent blue box.
- Whenever ϕ occurs at the end of a blue box, $\Box\phi$ may be put after that blue box.

Rules for □

Introduction of □:

$$\dfrac{\boxed{\begin{array}{c} \vdots \\ \phi \end{array}}}{\Box\phi}\,[\Box\ i]$$

Rules for □

Elimination of □:

$$\frac{\Box\phi}{\boxed{\begin{array}{c}\vdots \\ \phi \\ \vdots\end{array}}}\ [\Box\ e]$$

Extra Rules for KT45

$$\frac{\Box\phi}{\phi}\,[T] \qquad \frac{\Box\phi}{\Box\Box\phi}\,[4] \qquad \frac{\neg\Box\phi}{\Box\neg\Box\phi}\,[5]$$

Example Proof

$$\vdash_K \Box p \land \Box q \to \Box(p \land q)$$

1	$\Box p \land \Box q$	assumption
2	$\Box p$	$\land e_1$ 1
3	$\Box q$	$\land e_2$ 1
4	p	$\Box e$ 2
5	q	$\Box e$ 3
6	$p \land q$	$\land i$ 4,5
7	$\Box(p \land q)$	$\Box i$ 4–6
8	$\Box p \land \Box q \to \Box(p \land q)$	$\to i$ 1–7

Wise Women Puzzle

- Three wise women, each wearing one hat, among three available red hats and *two* available white hats
- Each wise woman is wise, can see other hats but not her own
- Queen asks first wise woman: Do you know the color of your hat.
  Answer: No
- Queen asks second wise woman: Do you know the color of your hat.
  Answer: No
- Queen asks third wise woman: Do you know the color of your hat?
- What is her answer?

Motivation

Reasoning about knowledge

We saw that KT45 can be used to reason about an agent's knowledge.

Difficulty

We have three agents (queen does not count), not just one. We want them to be able to reason about *each others* knowledge.

Idea

Introduce a $\Box$ operator for each agent, and a $\Box$ operator for a group of agents.

Modal Logic KT45^n

Agents

Assume a set $\mathcal{A} = \{1, 2, \ldots, n\}$ of agents.

Modal connectives

Replace $\Box$ by:

- K_i for each agent i
- E_G for any subset G of $\mathcal{A}$

Example

$K_1\ p \land K_1\ \neg\ K_2\ K_1\ p$ means:

Agent 1 knows p, and also that Agent 2 does not know that Agent 1 knows p.

Common Knowledge

“Everyone knows that everyone knows”

In $KT45^n$, $E_G\ E_G\ \phi$ is stronger than $E_G\ \phi$.

“Everyone knows everyone knows everyone knows”

In $KT45^n$, $E_G\ E_G\ E_G\ \phi$ is stronger than $E_G\ E_G\ \phi$.

Common knowledge

The infinite conjunction $E_G\phi \wedge E_G\ E_G\phi \wedge \ldots$ is called “common knowledge of ϕ”, denoted, $C_G\ \phi$.

Distributed Knowledge

Combine knowledge

If intelligent agents communicate with each other and use the knowledge each have, they can discover new knowledge.

Distributed knowledge

The operator $D_G\phi$ is called “distributed knowledge of ϕ”, denoted, $D_G\ \phi$.

Models of $\mathrm{KT45}^n$

Definition

A model $\mathcal{M} = (W, (R_i)_{i \in \mathcal{A}}, L)$ of the multi-modal logic $\mathrm{KT45}^n$ is specified by three things:

1. A set W, whose elements are called *worlds*;
2. For each $i \in \mathcal{A}$ a relation R_i on W, meaning $R_i \subseteq W \times W$, called the accessibility relations;
3. A labeling function $L : W \to \mathcal{P}(\mathtt{Atoms})$.

Semantics of $KT45^n$

Definition

Take a model $\mathcal{M} = (W, (R_i)_{i \in \mathcal{A}}, L)$ and a world $x \in W$. We define $x \Vdash \phi$ via structural induction:

- $x \Vdash p$ iff $p \in L(x)$
- $x \Vdash \neg\phi$ iff $x \nVdash \phi$
- $x \Vdash \phi \wedge \psi$ iff $x \Vdash \phi$ and $x \Vdash \psi$
- $x \Vdash \phi \vee \psi$ iff $x \Vdash \phi$ or $x \Vdash \psi$
- $x \Vdash \phi \rightarrow \psi$ iff $x \Vdash \psi$, whenever $x \Vdash \phi$
- ...

Semantics of $KT45^n$ (continued)

Definition

Take a model $\mathcal{M} = (W, (R_i)_{i \in \mathcal{A}}, L)$ and a world $x \in W$. We define $x \Vdash \phi$ via structural induction:

- ...
- $x \Vdash K_i\phi$ iff for each $y \in W$ with $R_i(x, y)$, we have $y \Vdash \phi$
- $x \Vdash E_G\phi$ iff for each $i \in G$, $x \Vdash K_i\phi$.
- $x \Vdash C_G\phi$ iff for each $k \geq 1$, we have $x \Vdash {E_G}^k\phi$.
- $x \Vdash D_G\phi$ iff for each $y \in W$, we have $y \Vdash \phi$, whenever $R_i(x, y)$ for all $i \in G$.

Formulation of Wise-Women Puzzle

Setup

- Wise woman i has red hat: p_i
- Wise woman i knows that wise woman j has a red hat: $K_i\ p_j$

Formulation of Wise-Women Puzzle

Initial situation

$$
\begin{aligned}
\Gamma \quad = \{ \quad & C(p_1 \vee p_2 \vee p_3), \\
& C(p_1 \rightarrow K_2 p_1), C(\neg p_1 \rightarrow K_2 \neg p_1), \\
& C(p_1 \rightarrow K_3 p_1), C(\neg p_1 \rightarrow K_3 \neg p_1), \\
& C(p_2 \rightarrow K_1 p_2), C(\neg p_2 \rightarrow K_1 \neg p_2), \\
& C(p_2 \rightarrow K_3 p_2), C(\neg p_2 \rightarrow K_3 \neg p_2), \\
& C(p_3 \rightarrow K_1 p_3), C(\neg p_2 \rightarrow K_1 \neg p_3), \\
& C(p_3 \rightarrow K_2 p_3), C(\neg p_2 \rightarrow K_2 \neg p_3)\}
\end{aligned}
$$

Announcements

First wise woman says "No"

$$C(\neg K_1 p_1 \wedge \neg K_1 \neg p_1)$$

Second wise woman says "No"

$$C(\neg K_2 p_2 \wedge \neg K_2 \neg p_2)$$

First Attempt

$$\Gamma, C(\neg K_1 p_1 \wedge \neg K_1 \neg p_1), C(\neg K_2 p_2 \wedge \neg K_2 \neg p_2) \vdash K_3 p_3$$

Problem

This does not take time into account. The second announcement can take the first announcement into account.

Solution

Prove separately:

Entailment 1 :

$$\Gamma, C(\neg K_1 p_1 \wedge \neg K_1 \neg p_1) \vdash C(p_2 \vee p_3)$$

Entailment 2 :

$$\Gamma, C(p_2 \vee p_3), C(\neg K_2 p_2 \wedge \neg K_2 \neg p_2) \vdash K_3 p_3$$

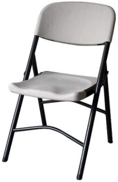
Proof

Through natural deduction in $KT45^n$.

Design Process

Constraints

Any design process is characterized by the management of constraints.



- cost
- storage space
- production process

Programming Language Design

Programs

Programs are instructions for a computer to perform a computation or *algorithm* and/or control devices such as disk drives and robots.

Purpose of programming languages

A programming language is a notation for writing programs.

Programming language design

Programming languages are designed by humans in order to meet the needs of a class of programming tasks.

Example: Java

Design goals for Java

1. simple, object oriented, and familiar
2. robust and secure
3. architecture neutral and portable
4. high performance
5. interpreted, threaded, and dynamic

Implicit design goal: Expressivity

It should be easy to write a wide variety of algorithms

Different Uses, Different Design Goals

English spoken in Lectures

Clarity and expressivity across a variety of cultural backgrounds

English used in Twitter

Brevity, “coolness”

Examples:

- ROFL
- POS

Programming Language Design

- Most programming languages are designed for humans to instruct computers
- Some languages are designed for *computers* to instruct computers. Example: PostScript

Example of PostScript Program

```
TeXDict begin/SDict 200 dict N SDict begin/@Special
/vs 792 N/ho 0 N/vo 0 N/hsc 1 N/vsc 1 N/ang 0 N/CLI
/rhiSeen false N/letter{}N/note{}N/a4{}N/legal{}N}B
/@hscale{@scaleunit div/hsc X}B/@vscale{@scaleunit
/hs X/CLIP 1 N}B/@vsize{/vs X/CLIP 1 N}B/@clip{/CLI
X}B/@voffset{/vo X}B/@angle{/ang X}B/@rwi{10 div/rw
/@rhi{10 div/rhi X/rhiSeen true N}B/@llx{/llx X}B/@
/urx X}B/@ury{/ury X}B/magscale true def end/@MacSe
{userdict/md get type/dicttype eq{userdict begin md
maxlength ge{/md md dup length 20 add dict copy def
/letter{}N/note{}N/legal{}N/od{txpose 1 0 mtx defau
atan/pa X newpath clippath mark{transform{itransfor
itransform lineto}}{6 -2 roll transform 6 -2 roll t
transform{itransform 6 2 roll itransform 6 2 roll i
```

Student Projects Written in PostScript

- LOH OEI HSIAN, CS3212 (2002) `http://www.comp.nus.edu.sg/~henz/lohoeihs.ps`
- Tan Woon Sern Elvin, CS3212 (2005) `http://www.comp.nus.edu.sg/~henz/tanwoons.ps`

Languages for Theory of Computation

Design Goals

- Expressivity
- Minimality

Example: Kepler's Laws and Newton's Laws

Kepler's Laws

- Planets move in ellipses
- Planet-sun line sweeps equal area during equal intervals
- P^2 is proportional to a^3

Example: Kepler's Laws and Newton's Laws

But why?

Keppler had his own ideas about this...

Example: Kepler's Laws and Newton's Laws

Newton's Laws of Motion

1. Bodies without force move at constant speed
2. Bodies with force experience acceleration: $F = ma$
3. To every action, there is an equal and opposite reaction

Newton's Law of Universal Gravitation

$F = G\frac{m_1 m_2}{r^2}$

Example: Newton's Laws and Kepler's Laws

Reduction of Kepler's Laws to Newton's Laws

Newton showed how to *derive* Kepler's Laws from his gravitation and motion laws.

Consequence

Kepler's laws are subordinate to Newton's laws. Physics will not change if we drop one of Kepler's laws.

Another Example of Minimalism in Theory

Peano axioms for natural numbers (and their equality):

1. For every natural number x, we have $x = x$
2. For all natural numbers x and y, if $x = y$, then $y = x$.
3. ...

There are nine Peano axioms that describe natural numbers *completely*.

Derived rule

$x > 1$ and $y > 1$ implies $x \cdot y > 1$

Status

Arithmetic will not change if we do not assume this derived rule.

Occam's Razor

In Latin

Entia non sunt multiplicanda praeter necessitatem.

In English

Entities must not be multiplied beyond necessity.

Reasons

- Practicality: easier to remember, document, etc
- Empical content: easier to reason about, to falsify
- Aesthetics: small theories are more beautiful

Programming: Theory vs Practice

For-loops vs while-loops

We can translate every for-loop into an equivalent while-loop.

```
for (i = 0; i < 10; i++) { print(i); }
```

becomes

```
i = 0; while (i < 10) { print(i); i = i + 1; }
```

So do we need for-loops?

Programmer's answer

Yes, please!
Having for-loops makes my programming so much easier!

Theoretician's answer

No!
If you add for-loops, I need to duplicate my proofs; many extra unnecessary pages!
Hours of work wasted!

A Theoretician's Programming Language

Do we need...

- **Function definition?** granted!
- **Function application?** granted!
- **Functions with multiple parameters?** no!
- **Numbers?** no!
- **Conditionals?** no!
- **Loops?** no!

Some Examples

```
function square(x) {
   return x * x;
}
square(13);
```

Do we need multiple arguments?

```
function plus(x,y) {
   return x + y;
}
plus(5,7);
```

Do we need multiple arguments?

```
function plus(x,y) {
   return x + y;
}
plus(5,7);
```

becomes

```
function plus(x) {
   function plusx(y) {
      return x + y;
   }
   return plusx;
}
var plusfive = plus(5);
plusfive(7);
```

Another Example

```
function power(x,y) {
    if (y === 0) return 1;
    return x * power(x,y-1);
}
power(2,4);
```

Do we need multiple arguments?

```
function power(x,y) {
   if (y === 0) return 1;
   return x * power(x,y-1);
}
power(2,4);
```

translates to:

```
function power(x) {
   return function(y) {
             if (y === 0) return 1;
             return x * power(x)(y-1);
          };
}
power(2)(4);
```

Do we need numbers?

Representing 0:

```
function zero(f) {
   return function(x) {
            return x;
          }
}
zero("something")("somethingelse")
```

Do we need numbers?

Representing 1:

```
function one(f) {
   return function(x) {
            return f(x);
          }
}
one(function(x) { return x*2; })(4)
```

Do we need numbers?

Representing 2:

```
function two(f) {
   return function(x) {
             return f(f(x));
          }
}
two(function(x) { return x*2; })(4)
```

Getting the number back

```
function two(f) {
   return function(x) {
              return f(f(x));
          }
}
function church2js(c) {
   return c(function(x) { return x+1; })(0);
}
church2js(two);
```

Multiplication

```
function times(x) {
   return function(y) {
             return function(f) {
                      return x(y(f));
                   }
             }
}
```

Multiplication

```
function three(f) {
   return function(x) {
             return f(f(f(x)));
          }
}
church2js(times(two)(three));
```

Conditionals

Conditional statements

```
if (20 < 10) { return 5; } else { return 7; }
```

Conditional expressions

```
(20 < 10) ? 5 : 7
```

Do we need conditionals?

Idea

Represent booleans with functions

The function "true"

```
function true(x) {
   return function(y) {
            return x;
   }
}
```

Do we need conditionals?

Idea

Represent booleans with functions

The function "false"

```
function false(x) {
   return function(y) {
            return y;
   }
}
```

Do we need conditionals?

Conditional in JavaScript

```
true ? 5 : 7;
```

Conditional using Encoding

```
true(5)(7);
```

Factorial in JavaScript

```
function factorial(x) {
    if (x === 0) return 1;
    return x * factorial(x - 1);
}
factorial(5);
```

Factorial using Conditional Expressions

```
function factorial(x) {
   return (x === 0) ? 1
          : x * factorial(x - 1);
}
factorial(5);
```

Step 1: Eliminate Recursive Call

```
function F(f) {
   return function(x) {
            return (x === 0) ? 1
                 : x * f(x - 1);
         };
}
```

Step 2: Find a Fix-Point Function

We need a function Y with the following properties:

$$Y(F) \equiv F(Y(F))$$

Step 2: Fix-Point Function

```
function Y(f) {
    return (function (x) {
                return f(function(y) {
                            return x(x)(y);
                        });
            })
            (function (x) {
                return f(function(y) {
                            return x(x)(y);
                        });
            });
}
```

Computing 5!

```
(function(f) {
    return (function (x) {
                return f(function(y) {
                          return x(x)(y);
                         }); })
           (function (x) {
                return f(function(y) {
                          return x(x)(y);
                         }); });
})(function(f) {
   return function(x) {
              return (x === 0) ? 1 : x * f(x - 1);
          };
})(5);
```

The Pure (Untyped) Lambda Calculus

As a sublanguage of JavaScript, the Lambda Calculus looks like this:

$$\mathbf{L} ::= x \mid (\mathbf{L})(\mathbf{L}); \mid \texttt{function}(x)\ \{\ \texttt{return}\ \mathbf{L};\ \}$$

Traditional Notation

As a sublanguage of JavaScript, the Lambda Calculus looks like this:

$$\mathbf{L} ::= x \mid (\mathbf{L}\ \mathbf{L}) \mid (\lambda\ x.L)\ \}$$

So: Why don't we program using the Lambda Calculus?

Answer

Other design goals are equally important!

Some design goals for full JavaScript

- Expressive
- Easy to learn
- Convenient to use

At the expense of...

simplicity!

Lambda Calculus: Some History

- Introduced by Alonzo Church in 1930s as a minimal formal system for recursion theory
- Later found to be equivalent to other computing frameworks (Church-Turing thesis)
- Used extensively in programming language theory and theoretical computer science

Summary

- Simplicity is an important and highly useful driving force behind science and engineering
- Enables insights that would otherwise remain lost in a thicket of details
- In practice, simplicity competes with other goals; keep it in mind when thinking about complex systems