Knowledge Representation

Lecture KRR-1

Introduction to the
Knowledge Represenation and
Reasoning Masters Course

Course Elements

- The course will cover the field of knowledge representation by giving a high-level overview of key aims and issues.
- Motivation and philosophical issues will be considered.
- Fundamental principles of logical analysis will be presented (concisely).
- Several important representational formalisms will be examined. Their motivation and capabilities will be explored.
- The potential practicality of KR methods will be illustrated by examining some examples of implemented systems.

Information and Learning

Course materials will be available from the module pages on Minerva and also at `teaching.bb-ai.net/KRR.html`.

There is no set text book for this course, but certain parts of the following provide very useful supporting material:

Russell S. and Norvig P. *Artificial Intelligence, A Modern Approach*, 3rd Edition (especially chapters 7–12).

Brachman RJ and Levesque HJ, *Knowledge Representation and Reasoning*, Morgan Kaufmann 2004

Poole D and Mackworth A, *Artificial intelligence: foundations of computational agents*,

There is an html version of this last title at
`http://artint.info/html/ArtInt.html`

Major Course Topics

- Classical Logic and Proof Systems.
- Automated Reasoning.
- Programming in *Prolog*.
- Representing and reasoning about time and change.
- Space and physical objects.
- Specialised AI representations: situation calculus, non-monotonic logic, description logic, fuzzy logic.
- Ontology and AI Knowledge Bases.

Coursework

The module will have four assessed pieces of work:

1. solution of problems by representing in logic and using an *automated theorem prover* (Prover9)
2. implementation of knowledge-based inference capabilities using *Prolog* (pairs work)
3. a short essay about a major challenge for KR&R (similar to the *Winograd Schema Challenge* (pairs work)
4. an *online test* consisting of short problems based on all the different reasoning systems covered in the module.

Relation to Basic Logical Background

A large amount of material is available in the form of slides and exercises.

We shall recap this but not revisit every detail.

We shall look at the application of KRR techniques in more general problem settings; and will often see that several representational formalisms and reasoning mechanisms need to be combined.

Knowledge Representation

Lecture KRR-2

Introduction to Knowledge Represenation and Reasoning

AI and the KR Paradigm

The methodology of Knowledge Representation and Automated Reasoning is one of the major strands of AI research.
It employs symbolic representation of information together with logical inference procedures as a means for solving problems.

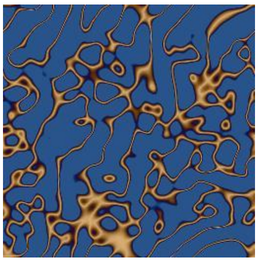
Although implementation and deployment of KRR techniques is very challenging, it has given rise to ideas and techniques that are used in a wide range of applications.

Most of the earliest investigations into AI adopted this approach and it is still going strong.
(It is sometimes called GOFAI — good old-fashioned AI.)

However, it is not the only (and perhaps not the most fashionable) approach to AI.

Neural Nets

One methodology for research in AI is to study the structure and function of the brain and try to recreate or simulate it.



How is intelligence dependent on its physical incarnation?

Situated and Reactive AI

Another approach is to tackle AI problems by observing and seeking to simulate intelligent behaviour by modelling the way in which an intelligent agent reacts to its environment.

A popular methodology is to look first at simple organisms, such as insects, as a first step towards understanding more high-level intelligence.

KRR *vs* ML

The view of AI taken in KRR is often considered to be opposed to that of Machine Learning.

This is partly true.

ML automatically creates models from data, that contain knowledge in an implicit form.
KRR typically uses hand-crafted models that store knowledge in an explicit way.

ML is primarily concerned with *classification*.
KRR is primarily concerned with *inference*.

Capabilities of ML systems are limited by the data upon which they are trained.
KRR can work in completely novel situations.

Intelligence *via* Language

The KR paradigm takes *language* as an essential vehicle for intelligence.

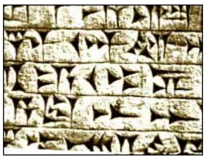
Animals can be seen as semi-intelligent because they only posses a rudimentary form of language.

The principle role of language is to *represent* information.

Language and Representation

Written language seems to have its origins in pictorial representations.

However, it evolved into a much more abstract representation.



Language and Logic

- Patters of natural language inference are used as a guide to the form of valid principles of logical deduction.
- Logical representations clean up natural language and aim to make it more definite.

For example:

If it is raining, I shall stay in.	$R \rightarrow S$
It is raining.	R
Therefore, I shall stay in.	$\therefore\ S$

Formalisation and Abstraction

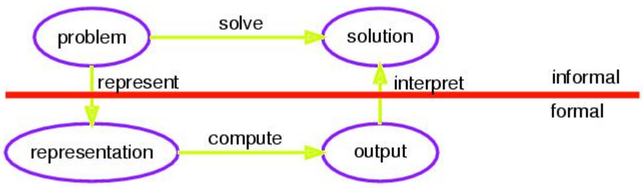
In employing a formal logical representation we aim to abstract from irrelevant details of natural descriptions to arrive at the essential structure of reasoning.

Typically we even ignore much of the logical structure present in natural language because we are only interested in (or only know how to handle) certain modes of reasoning.

For example, for many purposes we can ignore the tense structure of natural language.

Formal and Informal Reasoning

The relationship between formal and informal modes of reasoning might be pictured as follows:



Reasoning in natural language can be regarded as *semi-formal*.

What do we represent?

- Our problem.
- What would count as a solution.
- Facts about the world.
- Logical properties of abstract concepts (i.e. how they can take part in inferences).
- Rules of inference.

Finding a “Good” Representation

- We must determine what knowledge is relevant to the problem.
- We need to find a suitable *level of abstraction*.
- Need a representation language in which problem and solution can be adequately expressed.
- Need a *correct* formalisation of problem and solution in that language.
- We need a *logical theory* of the modes of reasoning required to solve the problem.

Inference and Computation

A tough issue that any AI reasoning system must confront is that of *Tractability*.

A problem domain is *intractable* if it is not possible for a (conventional) computer program to solve it in 'reasonable' time (and with 'reasonable' use of other resources such as memory).

Certain classes of logical problem are not only intractable but also *undecidable*.
This means that there is no program that, given any instance of the problem, will in *finite time* either: a) find a solution; or b) terminate having determined that no solution exists.

Later in the course we shall make these concepts more precise.

Time and Change

$1+1=2$ Standard, classical logic was developed primarily for applications to mathematics. $1+1=2$

Since mathematical truths are eternal, it is not geared towards representing temporal information.

However, time and change play an essential role in many AI problem domains. Hence, formalisms for temporal reasoning abound in the AI literature.

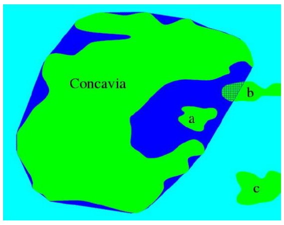
We shall study several of these and the difficulties that obstruct any simple approach (in particular the famous *Frame Problem*).

Spatial Information

Knowledge of spatial properties and relationships is required for many commonsense reasoning problems.

While mathematical models exist they are not always well-suited for AI problem domains.

We shall look at some ways of representing qualitative spatial information.



Describing and Classifying Objects

To solve simple commonsense problems we often need detailed knowledge about everyday objects.

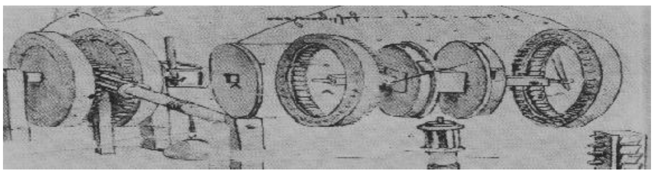
Can we precisely specify the properties of type of object such as a cup?

Which properties are essential?

Combining Space and Time

For many purposes we would like to be able to reason with knowledge involving both spatial and temporal information.

For example we may want to reason about the working of some physical mechanism:



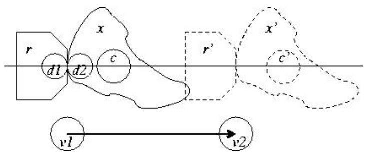
Robotic Control

An important application for spatio-temporal reasoning is robot control.

Many AI techniques (as well as a great deal of engineering technology) have been applied to this domain.

While success has been achieved for some constrained envioronments, flexible solutions are elusive.

Versatile high-level control of autonomous agents is a major goal of KR.



Uncertainty

Much of the information available to an intelligent (human or computer) is affected by some degree of uncertainty.

This can arise from: unreliable information sources, inaccurate measurements, out of date information, unsound (but perhaps potentially useful) deductions.

This is a big problem for AI and has attracted much attention. Popular approaches include *probabalistic* and *fuzzy* logics.

But ordinary classical logics can mitigate the problem by use of *generality*. E.g. instead of prob$(\phi) = 0.7$, we might assert a more general claim $\phi \vee \psi$.

Ontology

Literally *Ontology* means the study of *what exists*.
It is studied in philosophy as a branch of *Metaphysics*.

In KR the term Ontology is used to refer to a rigorous logical specification of a domain of objects and the concepts and relationships that apply to that domain.

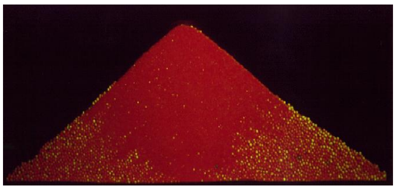
Ontologies are intended to guarantee the coherence of information and to allow relyable exchange of information between computer systems.

Use of ontologies is one of the main ways in which KRR techniques are exploited in modern software applications.

Issues of Ambiguity and Vagueness

A huge problem that obstructs the construction of rigorous ontologies is the widespread presence of *ambiguity* and *vagueness* in natural concepts.

For example: tall, good, red, cup, mountain.



How many grains make a *heap*?

iv

v

vi

Knowledge Representation

Lecture KRR-3

Classical Logic I:
Concepts and Uses of Logic

Lecture Plan

- Formal Analysis of Inference
- Propositional Logic
- Validity
- Quantification
- Uses of Logic

Logical Form

A *form* of an object is a structure or pattern which it exhibits.

A *logical form* of a linguistic expression is an aspect of its structure which is relevant to its behaviour with respect to inference.

To illustrate this we consider a mode of inference which has been recognised since ancient times.

Logical Form of an Argument

If Leeds is in Yorkshire then Leeds is in the UK
Leeds is in Yorkshire

Therefore, Leeds is in the UK

$$\begin{array}{l} \text{If } P \text{ then } Q \\ P \\ \hline \therefore\ Q \end{array} \qquad\qquad \begin{array}{l} P \rightarrow Q \\ P \\ \hline Q \end{array}$$

(The Romans called this type of inference *modus ponendo ponens*.)

Propositions

The preceding argument can be explained in terms of *propositional* logic.

A proposition is an expression of a fact.

The symbols, P and Q, represent propositions and the logical symbol ' $\rightarrow$ ' is called a propositional connective.

Many systems of propositional logic have been developed. In this lecture we are studying *classical* — i.e. the best established — propositional logic.

In classical propositional logic it is taken as a principle that:

Every proposition is either *true* or *false* and not both.

Complex Propositions and Connectives

Propositional logic deals with inferences governed by the meanings of propositional *connectives*. These are expressions which operate on one or more propositions to produce another more complex proposition.

The connectives dealt with in standard propositional logic correspond to the natural language constructs:

- '... and ...',
- '... or ...'
- 'it is not the case that...'
- 'if ... then ...'.

Symbols for the Connectives

The propositional connectives are represented by the following symbols:

and	$\wedge$	$(P \wedge Q)$
or	$\vee$	$(P \vee Q)$
if ... then	$\rightarrow$	$(P \rightarrow Q)$
not	$\neg$	$\neg P$

More complex examples:

$((P \wedge Q) \vee R)$, $(\neg P \rightarrow \neg(Q \vee R))$

Brackets prevent ambiguity which would otherwise occur in a formula such as '$P \wedge Q \vee R$'.

Propositional Formulae

We can precisely specify the well-formed formulae of propositional logic by the following (recursive) characterisation:

- Each of a set, $\mathcal{P}$, of propositional constants P_i is a formula.
- If α is a formula so is $\neg\alpha$.
- If α and β are formulae so is $(\alpha \wedge \beta)$.
- If α and β are formulae so is $(\alpha \vee \beta)$.

The propositional *connectives* $\neg$, $\wedge$ and $\vee$ are respectively called *negation*, *conjunction* and *disjunction*.

Proposition Symbols and Schematic Variables

The symbols P, Q etc. occurring in propositional formulae should be understood as abbreviations for actual propositions such as 'It is Tuesday' and 'I am bored'.

In defining the class of propositional formulae I used Greek letters (α and β) to stand for arbitrary propositional formulae. These are called *schematic variables*.

Schematic variables are used to refer classes of expression sharing a common form. Thus they can be used for describing patterns of inference.

Inference Rules

An inference rule characterises a pattern of valid deduction.

In other words, it tells us that if we accept as true a number of propositions — called premisses — which match certain patterns, we can deduce that some further proposition is true — this is called the conclusion.

Thus we saw that from two propositions with the forms $\alpha \rightarrow \beta$ and α we can deduce β.
The inference from $P \rightarrow Q$ and P to Q is of this form.

An inference rule can be regarded as a special type of re-write rule: one that preserves the truth of formulae — i.e. if the premisses are true so is the conclusion.

More Simple Examples

'And' Elimination

$$\frac{\alpha \wedge \beta}{\alpha} \qquad \frac{\alpha \wedge \beta}{\beta}$$

'And' Introduction

$$\frac{\alpha \qquad \beta}{\alpha \wedge \beta}$$

'Or' Introduction

$$\frac{\alpha}{\alpha \vee \beta} \qquad \frac{\alpha}{\beta \vee \alpha}$$

'Not' Elimination

$$\frac{\neg\neg\alpha}{\alpha}$$

Logical Arguments and Proofs

A logical *argument* consists of a set of propositions $\{P_1, \ldots, P_n\}$ called premisses and a further proposition C, the conclusion.

Notice that in speaking of an argument we are not concerned with any sequence of inferences by which the conclusion is shown to follow from the premisses. Such a sequence is called a *proof*.

A set of inference rules constitutes a *proof system*.

Inference rules specify a class of primitive arguments which are justified by a single inference rule. All other arguments require proof by a series of inference steps.

A 2-Step Proof

Suppose we know that 'If it is Tuesday or it is raining John stays in bed all day' then if we also know that 'It is Tuesday' we can conclude that 'John is in Bed'.

Using T, R and B to stand for the propositions involved, this conclusion could be proved in propositional logic as follows:

$$\frac{((T \vee R) \rightarrow B) \qquad \dfrac{T}{(T \vee R)}}{B}$$

Here we have used the 'or introduction' rule followed by good old *modus ponens*.

Provability

To assert that C can be proved from premisses $\{P_1, \ldots, P_n\}$ in a proof system S we write:

$$P_1, \ldots, P_n \vdash_S C$$

This means that C can be derived from the formulae $\{P_1, \ldots, P_n\}$ by a series of inference rules in the proof system S.

When it is clear what system is being used we may omit the subscript S on the '$\vdash$' symbol.

Validity

An argument is called *valid* if its conclusion is a consequence of its premisses. Otherwise it is invalid. This needs to be made more precise:

One definition of validity is: An argument is valid if it is not possible for its premisses to be true and its conclusion is false.

Another is: in all possible circumstances in which the premisses are true, the conclusion is also true.

To assert that the argument from premisses $\{P_1, \ldots, P_n\}$ to conclusion C is valid we write:

$$P_1, \ldots, P_n \models C$$

Provability *vs* Validity

We have defined **provability** as a property of an argument which depends on the inference rules of a logical proof system.

Validity on the other hand is defined by appealing directly to the meanings of formulae and to the circumstances in which they are true or false.

In the next lecture we shall look in more detail at the relationship between validity and provability. This relationship is of central importance in the study of logic.

To characterise validity we shall need some precise specification of the 'meanings' of logical formulae. Such a specification is called a *formal semantics*.

Relations

In propositional logic the smallest meaningful expression that can be represented is the proposition. However, even atomic propositions (those not involving any propositional connectives) have internal logical structure.

In *predicate* logic atomic propositions are analysed as consisting of a number of *individual constants* (i.e. names of objects) and a *predicate*, which expresses a *relation* between the objects.

$$\mathsf{R}(\mathsf{a}, \mathsf{b}, \mathsf{c}) \qquad \mathsf{Loves}(\mathsf{john}, \mathsf{mary})$$

With many binary (2-place) relations the relation symbol is often written between its operands — e.g. $4 > 3$.

Unary (1-place) relations are also called *properties* — $\mathsf{Tall}(\mathsf{tom})$.

Universal Quantification

Useful information often takes the form of statements of general property of entities. For instance, we may know that 'every dog is a mammal'. Such facts are represented in predicate logic by means of *universal quantification*.

Given a complex formula such as $(\mathsf{Dog}(\mathsf{spot}) \rightarrow \mathsf{Mammal}(\mathsf{spot}))$, if we remove one or more instances of some individual constant we obtain an incomplete expression $(\mathsf{Dog}(\ldots) \rightarrow \mathsf{Mammal}(\ldots))$, which represents a (complex) property.

To assert that this property holds of all entities we write:

$$\forall x[\mathsf{Dog}(x) \rightarrow \mathsf{Mammal}(x)]$$

in which '$\forall$' is the universal quantifier symbol and x is a *variable* indicating which property is being quantified.

An Argument Involving Quantification

An argument such as:

Everything in Yorkshire is in the UK
Leeds is in Yorkshire

Therefore Leeds is in the UK

can now be represented as follows:

$\forall x[\text{Inys}(x) \rightarrow \text{Inuk}(x)]$
$\text{Inys}(l)$

$\text{Inuk}(l)$

Later we shall examine quantification in more detail.

Uses of Logic

Logic has always been important in philosophy and in the foundations of mathematics and science. Here logic plays a foundational role: it can be used to check consistency and other basic properties of precisely formulated theories.

In computer science, logic can also play this role — it can be used to establish general principles of computation; but it can also play a rather different role as a 'component' of computer software: computers can be programmed to carry out logical deductions. Such programs are called *Automated Reasoning* systems.

Formal Specification of Hardware and Software

Since logical languages provide a flexible but very precise means of description, they can be used as specification language for computer hardware and software.

A number of tools have been developed which help developers go from a formal specification of a system to an implementation.

However, it must be realised that although a system may completely satisfy a formal specification it may still not behave as intended — there may be errors in the formal specification.

Formal Verification

As well as being used for specifying hardware or software systems, descriptions can be used to verify properties of systems.

If Θ is a set of formulae describing a computer system and π is a formula expressing a property of the system that we wish to ensure (eg. π might be the formula $\forall x[\text{Employee}(x) \rightarrow \text{age}(x) > 0]$), then we must verify that:

$$\Theta \models \pi$$

We can do this using a proof system S if we can show that:

$$\Theta \vdash_S \pi$$

Logical Databases

A set of logical formulae can be regarded as a *database*.

A logical database can be *queried* in a very flexible way, since for any formula ϕ, the semantics of the logic precisely specify the conditions under which ϕ is a consequence of the formulae in the database.

Often we may not only want to know whether a proposition is true but to find all those entities for which a particular formula containing variables holds.
e.g.

query: $\text{Located}(x, y) \;\wedge \text{Furniture}(x)$?
Ans: $\langle x = \text{sofa1}, y = \text{lounge} \rangle$, $\langle x = \text{table1}, y = \text{kitchen} \rangle, \ldots$

Logic and Intelligence

The ability to reason and draw consequences from diverse information may be regarded as fundamental to *intelligence*.

As the principal intention in constructing a logical language is to precisely specify correct modes of reasoning, a logical system (i.e. a logical language plus some proof system) might in itself be regarded as a form of *Artificial Intelligence*.

However, as we shall see as this course progresses, there are many obstacles that stand in the way of achieving an 'intelligent' reasoning system based on logic.

Knowledge Representation

Lecture KRR-4

Classical Logic II:
Formal Systems, Proofs and Semantics

Sequents

A sequent is an expression of the form:

$$\alpha_1, \ \ldots, \ \alpha_m \Rightarrow \ \beta_1, \ \ldots, \ \beta_n$$

(where the αs and βs are logical formulae).

This asserts that:

If all of the αs are true then at least one of the βs is true.

Hence, it means the same as:

$$(\alpha_1 \wedge \ \ldots \ \wedge \alpha_m) \ \rightarrow \ (\beta_1 \vee \ \ldots \ \vee \beta_n)$$

This notation — as we shall soon see — is very useful in presenting inference rules in a concise and uniform way.

Notation Issues

Many articles and textbooks write sequents using the notation:

$$\alpha_1, \ \ldots, \ \alpha_m \vdash \ \beta_1, \ \ldots, \ \beta_n$$

instead of $\alpha_1, \ \ldots, \ \alpha_m \Rightarrow \ \beta_1, \ \ldots, \ \beta_n$.

(I used that notation in previous versions of my slides and notes.)

However, I believe this is confusing because *the $\vdash$ symbol normally means provability*. Indeed, some authors do talk about the '$\vdash$' in a sequent as though it refers to a provability relation. But this is wrong: it functions as a special kind of connective.

The sequent calculus was orignated by Gerhard Gentzen, who used '$\rightarrow$' in his sequents (and used '$\supset$ for the implication symbol).

Special forms of sequent

A sequent with an empty left-hand side:

$$\Rightarrow \ \beta_1, \ \ldots, \ \beta_n$$

asserts that at least one of the βs must be true without assuming any premisses to be true.

If the simple sequent $\Rightarrow \beta$ is valid, then β is called a *logical theorem*.

A sequent with an empty right-hand side: $\alpha_1, \ \ldots, \ \alpha_m \Rightarrow$

asserts that the set of premisses $\{\alpha_1, \ \ldots, \ \alpha_m\}$ is *inconsistent*.

Sequent Calculus Inference Rules

A sequent calculus inference rule specifies a pattern of reasoning in terms of sequents rather than formulae.

Eg. a sequent calculus 'and introduction' is specified by:

$$\frac{\Gamma \Rightarrow \alpha, \ \Delta \quad \textit{and} \quad \Gamma \Rightarrow \beta, \ \Delta}{\Gamma \Rightarrow (\alpha \wedge \beta), \ \Delta} [\Rightarrow \ \wedge]$$

where Γ and Δ are any series of formulae.

In a sequent calculus we also have rules which introduce symbols into the premisses:

$$\frac{\alpha, \ \beta, \ \Gamma \Rightarrow \ \Delta}{(\alpha \wedge \beta), \ \Gamma \Rightarrow \ \Delta} [\wedge \Rightarrow]$$

Ordering Does Not Matter

The applicability of a rule to a formula depends on which side of the $\Rightarrow$ it occurs on.
But the ordering of formulae on the same side does not matter.

Thus each rule can apply to either any formula on the left or any formula on the right.

Hence, sequent calculus rules normally come in pairs, with one being applicable to a certain kind of formula occurring on the left (e.g. $[\Rightarrow \ \wedge]$) and the other applicable when that kind of formula occurs on the right (e.g. $[\wedge \Rightarrow]$).

(We shall see this in the proof system that will be presented shortly.)

Sequent Calculus Proof Systems

To assert that a sequent is provable in a sequent calculus system, SC, I shall write:

$$\vdash_{\mathsf{SC}} \alpha_1,\ \ldots,\ \alpha_m \Rightarrow\ \beta_1,\ \ldots,\ \beta_n$$

Construing a proof system in terms of the provability of sequents allows for much more uniform presentation than can be given in terms of provability of conclusions from premisses.

We start by stipulating that all sequents of the form

$$\alpha,\ \Gamma \Rightarrow\ \alpha,\ \Delta$$

are immediately provable.

We then specify how each logical symbol can be introduced into the left and right sides of a sequent (see next slide).

A Propositional Sequent Calculus

Rules:

$$\frac{\textbf{Axiom}}{\alpha, \Gamma \Rightarrow \alpha, \Delta}$$

$$\frac{\alpha,\ \beta,\ \Gamma \Rightarrow\ \Delta}{(\alpha \wedge \beta),\ \Gamma \Rightarrow\ \Delta}[\wedge\Rightarrow] \qquad \frac{\Gamma \Rightarrow \alpha, \Delta \quad \textit{and} \quad \Gamma \Rightarrow \beta, \Delta}{\Gamma \Rightarrow (\alpha \wedge \beta),\ \Delta}[\Rightarrow\wedge]$$

$$\frac{\alpha, \Gamma \Rightarrow \Delta \quad \textit{and} \quad \beta, \Gamma \Rightarrow \Delta}{(\alpha \vee \beta),\ \Gamma \Rightarrow\ \Delta}[\vee\Rightarrow] \qquad \frac{\Gamma \Rightarrow\ \alpha,\ \beta,\ \Delta}{\Gamma \Rightarrow\ (\alpha \vee \beta),\ \Delta}[\Rightarrow\vee]$$

$$\frac{\Gamma \Rightarrow\ \alpha,\ \Delta}{\neg\alpha,\ \Gamma \Rightarrow\ \Delta}[\neg\Rightarrow] \qquad \frac{\Gamma,\ \alpha \Rightarrow\ \Delta}{\Gamma \Rightarrow\ \neg\alpha,\ \Delta}[\Rightarrow\neg]$$

Re-Write Rules

Re-write rules, are an easy way to specify transformations of a formula (on either side of a sequent) to an equivalent formula.

We shall write: $\alpha \to \beta \Longrightarrow \neg\alpha \vee \beta\ [\to r.w]$
as a short specification for the rules:

$$\frac{\neg\alpha \vee \beta,\ \Gamma \Rightarrow\ \Delta}{\alpha \to \beta,\ \Gamma \Rightarrow\ \Delta}[\to r.w.] \qquad \frac{\Gamma \Rightarrow\ \neg\alpha \vee \beta,\ \Delta}{\Gamma \Rightarrow\ \alpha \to \beta,\ \Delta}[\to r.w.]$$

Exercises:
1. Show that the rules [$\vee \Rightarrow$] and [$\Rightarrow\ \vee$] can be replaced by the rewrite rule $\alpha \vee \beta \ \Longrightarrow\ \neg(\neg\alpha \wedge \neg\beta)$

2. Specify rules for [$\to \Rightarrow$] and [$\Rightarrow\ \to$] that directly eliminate the $\to$ connective without replacing it by an equivalent form.

Sequent Calculus Proofs

The beauty of the sequent calculus system is its *reversibility*.

To test whether a sequent, $\Gamma \Rightarrow \Delta$, is provable we simply apply the symbol introduction rules backwards. Each time we apply a rule, one connective is eliminated. With some rules two sequents then have to be proved (the proof branches) but eventually every branch will terminate in a sequent containing only atomic propositions. If all these final sequents are axioms, then $\Gamma \Rightarrow \Delta$ is proved, otherwise it is not provable.

Note that the propositional sequent calculus rules can be applied in any order.

This calculus is easy to implement in a computer program.

Proof Example 1

$$\overline{(P \to Q),\ P \Rightarrow\ Q}$$

Proof Example 2

$$\overline{\neg(P \wedge \neg Q) \Rightarrow\ (P \to Q)}$$

Proof Example 3

$$\overline{((P \vee Q) \vee R),\ (\neg P \vee S),\ \neg(Q \wedge \neg S) \Rightarrow (R \vee S)}$$

Formal Semantics

We have seen that a notion of *validity* can be defined independently of the notion of *provability*:

An argument is valid if it is not possible for its premisses to be true and its conclusion is false.

We could make this precise if we could somehow specify the conditions under which a logical formulae is true.

Such a specification is called a *formal semantics* or an *interpretation* for a logical language.

Interpretation of Propositional Calculus

To specify a formal semantics for propositional calculus we take literally the idea that 'a proposition is either true or false'.

We say that the *semantic value* of every propositional formula is one of the two values **T** or **F** — called *truth-values*.

For the atomic propositions this value will depend on the particular fact that the proposition asserts and whether this is true. Since propositional logic does not further analyse atomic propositions we must simply assume there is some way of determining the truth values of these propositions.

The connectives are then interpreted as *truth-functions* which completely determine the truth-values of complex propositions in terms of the values of their atomic constituents.

Truth-Tables

The truth-functions corresponding to the propositional connectives $\neg$, $\wedge$ and $\vee$ can be defined by the following tables:

α	$\neg\alpha$
F	**T**
T	**F**

α	β	$(\alpha \wedge \beta)$
F	**F**	**F**
F	**T**	**F**
T	**F**	**F**
T	**T**	**T**

α	β	$(\alpha \vee \beta)$
F	**F**	**F**
F	**T**	**T**
T	**F**	**T**
T	**T**	**T**

These give the truth-value of the complex proposition formed by the connective for all possible truth-values of the component propositions.

The Truth-Function for '$\rightarrow$'

The truth-function for ' $\rightarrow$ ' is defined so that a formulae $(\alpha \rightarrow \beta)$ is always true except when α is true and β is false:

α	β	$(\alpha \rightarrow \beta)$
F	**F**	**T**
F	**T**	**T**
T	**F**	**F**
T	**T**	**T**

So the statement 'If logic is interesting then pigs can fly' is true if either 'Logic is interesting' is false or 'Pigs can fly is true'.

Thus a formula $(\alpha \rightarrow \beta)$ is *truth-functionally equivalent* to $(\neg\alpha \vee \beta)$.

Propositional Models

A propositional *model* for a propositional calculus in which the propositions are denoted by the symbols $P_1, \ldots, P_n$, is a specification assigning a truth-value to each of these proposition symbols. It might by represented by, e.g.:

$$\{\langle P_1 = \mathbf{T}\rangle, \langle P_2 = \mathbf{F}\rangle, \langle P_3 = \mathbf{F}\rangle, \langle P_4 = \mathbf{T}\rangle, \ldots\}$$

Such a model determines the truth of all propositions built up from the atomic propositions $P_1, \ldots, P_n$. (The truth-value of the atoms is given directly and the values of complex formulae are determined by the truth-functions.)

If a model, $\mathcal{M}$, makes a formula, ϕ, true then we say that $\mathcal{M}$ *satisfies* ϕ.

Validity in terms of Models

Recall that an argument's being valid means that: in all possible circumstances in which the premisses are true the conclusion is also true.

From the point of view of truth-functional semantics each model represents a possible circumstance — i.e. a possible set of truth values for the atomic propositions.

To assert that an argument is *truth-functionally valid* we write

$$P_1,\ \ldots,\ P_n\ \models_{TF}\ C$$

and we define this to mean that ALL models which satisfy ALL of the premisses, $P_1,\ \ldots,\ P_n$ also satisfy the conclusion C.

Soundness and Completeness

A proof system is *complete* with respect to a formal semantics if every argument which is valid according to the semantics is also provable using the proof system.

A proof system is *sound* with respect to a formal semantics if every argument which is provable with the system is also valid according to the semantics.

It can be shown that the system of sequent calculus rules, SC, is both sound and complete with respect to the truth-functional semantics for propositional formulae.

Thus, $\vdash_{\text{SC}}\ \Gamma \vdash C$ if and only if $\Gamma\ \models_{TF}\ C$.

(How this can be show is beyond the scope of this course.)

More about Quantifiers

We shall now look again at the notation for expressing quantification and what it means.

First suppose, $\phi(\ldots)$ expresses a property — i.e. it is a predicate logic formulae with (one or more occurrences of) a name removed.

If we want say that something exists which has this property we write:

$$\exists x[\phi(x)]$$

'$\exists$' being the *existential quantifier* symbol.

Multiple Quantifiers

Consider the sentence: 'Everybody loves somebody' We can consider this as being formed from an expression of the form loves(john, mary) by the following stages.

First we remove mary to form the property loves(john, ...) which we existentially quantify to get: $\exists x[$ loves(john, x)$]$

Then by removing john we get the property $\exists x[$ loves($\ldots, x$)$]$ which we quantify universally to end up with:

$$\forall y[\ \exists x[\text{loves}(y, x)]\ ]$$

Notice that each time we introduce a new quantifier we must use a new variable letter so we can tell what property is being quantified.

Defining $\exists$ in Terms of $\forall$

We shall shortly look at sequent rules for handling the universal quantifier.

Predicate logic formulae will in general contain both universal ($\forall$) and existential ($\exists$) quantifiers. However, in the same way that in propositional logic we saw that $(\alpha \rightarrow \beta)$ can be replaced by $(\neg\alpha \vee \beta)$, the existential quantifier can be eliminated by the following re-write rule.

$$\exists v[\phi(v)] \qquad \Longrightarrow \qquad \neg\forall v[\neg\phi(v)]$$

These two forms of formula are equivalent in meaning.

The Sequent Rule for $\Rightarrow \forall$

$$\frac{\Gamma \Rightarrow\ \phi(\kappa),\ \Delta}{\Gamma \Rightarrow\ \forall v[\phi(v)],\ \Delta}\,[\Rightarrow \forall]^{*}$$

The * indicates a special condition:

The constant κ must not occur anywhere else in the sequent.

This restriction is needed because if κ occurred in another formulae of the sequent then it could be that $\phi(\kappa)$ is a consequence which can only be proved in the special case of κ.

On the other hand if κ is not mentioned elsewhere it can be regarded as an *arbitrary* object with no special properties.
If the property $\phi(\ldots)$ can be proven true of an arbitrary object it must be true of all objects.

An example

As an example we now prove that the formula $\forall x[P(x) \vee \neg P(x)]$ is a theorem of predicate logic. This formula asserts that every object either has or does not have the property $P(\ldots)$.

$$\cfrac{\cfrac{\cfrac{P(a) \Rightarrow\ P(a)}{\Rightarrow\ P(a),\ \neg P(a)}\,[\Rightarrow \neg]}{\Rightarrow\ (P(a) \vee \neg P(a))}\,[\Rightarrow\ \vee\,]}{\Rightarrow\ \forall x[P(x) \vee \neg P(x)]}\,[\Rightarrow \forall]$$

Here the (reverse) application of the $[\Rightarrow \forall]$ rule could have been used to introduce not only a but any name, since no names occur on the LHS of the sequent.

Another Example

Consider the follwing illegal application of $[\Rightarrow \forall]$:

$$\frac{P(b) \Rightarrow\ P(b)]}{P(b) \Rightarrow\ \forall x[P(x)]}\,[\Rightarrow \forall]^{\dagger}$$

† This is an incorrect application of the rule, since b already occurs on the LHS of the sequent.

(Just because b has the property $P(\ldots)$ we cannot conclude that everything has this property.)

A Sequent Rule for $\forall\Rightarrow$

A formula of the form $\forall\upsilon[\phi(\upsilon)]$ clearly entails $\phi(\kappa)$ for any name κ. Hence the following sequent rule clearly preserves validity:

$$\frac{\phi(\kappa),\ \Gamma \Rightarrow\ \Delta}{\forall\upsilon[\phi(\upsilon)],\ \Gamma \Rightarrow\ \Delta}\,[\forall\Rightarrow\,]$$

But, the formulae $\phi(\kappa)$ makes a much weaker claim than $\forall\upsilon[\phi(\upsilon)]$. This means that this rule is not *reversible* since, the bottom sequent may be valid but not the top one.

Consider the case:

$$\frac{F(a) \Rightarrow\ (F(a) \wedge F(b))}{\forall x[F(x)] \Rightarrow\ (F(a) \wedge F(b))}\,[\forall\Rightarrow\,]$$

A Reversible Version

A quantified formula $\forall\upsilon[\phi(\upsilon)]$ has as consequences all formulae of the form $\phi(\kappa)$; and, in proving a sequent involving a universal premiss, we may need to employ many of these *instances*.

A simple way of allowing this is by using the following rule:

$$\frac{\phi(\kappa),\ \forall\upsilon[\phi(\upsilon)],\ \Gamma \Rightarrow\ \Delta}{\forall\upsilon[\phi(\upsilon)],\ \Gamma \Rightarrow\ \Delta}\,[\forall\Rightarrow\,]$$

When applying this rule backwards to test a sequent we find a universal formulae on the LHS and add some instance of this formula to the LHS.

Note that the universal formula is not removed because we may later need to apply the rule again to add a different instance.

An Example Needing 2 Instantiations

We can now see how the sequent we considered earlier can be proved by applying the $[\forall\Rightarrow\,]$ rule twice, to instantiate the same universally quantified property with two different names.

$$\cfrac{\cfrac{\cfrac{F(a),\ldots\Rightarrow F(a) \quad and \quad \ldots, F(b), \ldots\Rightarrow F(b)}{F(a),\ F(b), \forall x[F(x)] \Rightarrow\ (F(a) \wedge F(b))}\,[\Rightarrow \wedge]}{F(a),\ \forall x[F(x)] \Rightarrow\ (F(a) \wedge F(b))}\,[\forall\Rightarrow\,]}{\forall x[F(x)] \Rightarrow\ (F(a) \wedge F(b))}\,[\forall\Rightarrow\,]$$

Termination Problem

We now have the problem that the (reverse) application of $[\forall\Rightarrow\,]$ results in a more complex rather than a simpler sequent.

Furthermore, in any application of $[\forall\Rightarrow\,]$ we must choose one of (infinitely) many names to instantiate.

Although there are various clever things that we can do to pick instances that are likely to lead to a proof, these problems are fundamentally insurmountable.

This means that unlike with propositional sequent calculus, there is no general purpose automaitc procedure for testing the validity of sequents containing quantified formulae.

Decision Procedures

A *decision procedure* for some class of problems is an algorithm which can solve any problem in that class in a finite time (i.e. by means of a finite number of computational steps).

Generally we will be interested in some infinite class of similar problems such as:

1. problems of adding any two integers together
2. problems of solving any polynomial equation
3. problems of testing validity of any propositional logic sequent
4. problems of testing validity of any predicate logic sequent

Decidability

A class of problems is *decidable* if there is a decision procedure for that class; otherwise it is *undecidable*.

Problem classes 1–3 of the previous slide are decidable, whereas class 4 is known to be undecidable.

Undecidability of testing validity of entailments in a logical language is clearly a major problem if the language is to be used in a computer system: a function call to a procedure used to test entailments will not necessarily terminate.

Semi-Decidability

Despite the fact that predicate logic is undecidable, the rules that we have given for the quantifiers to give us a *complete* proof system for predicate logic.

Furthermore, it is even possible to devise a strategy for picking instants in applying the $[\forall\Rightarrow]$ rule, such that every *valid* sequent is provable in finite time.

However, there is no procedure that will demonstrate the *invalidity* of every *invalid* sequent in finite time.

A problem class, where we want a result *Yes* or *No* for each problem, is called *(positively) semi-decidable* if every positive case can be verified in finite time but there is no procedure which will refute every negative case in finite time.

iv

v

vi

Knowledge Representation

Lecture KRR-5

Classical Logic III:
Representation in First-Order Logic

First-Order Logic

As we have seen First-Order Logic extends Propositional Logic in two ways:

- The meanings of 'atomic' propositions may be represented in terms of properties and relations holding among named objects.
- Expressive power is further increased by the use of variables and 'quantifiers', which can be used to represent generalised or non-specific information.

(Note: a quantifier is called *first-order* if its variable ranges over individual entities of a domain. *Second-order* quantification is where the quantified variable ranges over *sets* of entities. In this course we shall restrict our attention to the first-order case.)

Terminology

The terms *predicate*, *relation*, and *property* are more or less equivalent.

'Property' tends to imply a predicate with exactly one argument (e.g. $P(x)$, Red(x), Cat(x)).

'Relation' tends to imply a predicate with at least two arguments (e.g. $R(x, y)$, Taller(x, y), Gave(x, y, z)).

The term 'Predicate' does not usually imply anything about the number of arguments (athough occasionally it is used to imply just one argument).

(First-Order Logic is sometimes referred to as 'Predicate Logic'.)

Symbols of First-Order Logic

First-order logic employs the following symbols:

- Predicate symbols each with a fixed *arity* (i.e. number of arguments): P, Q, R, Red, Taller ...
- Constants (names of particular individuals): a, b, john, leeds, ...
- Variable symbols: x, y, z, u, v, ...
- (Truth-Functional) Connectives — unary: $\neg$, binary: $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$
- Quantifiers: $\forall$, $\exists$
- The equality relation: $=$ (First-Order logic may be used with or without equality.)

Formulae of First-Order Logic

An *atomic* formula is an expression of the form:

$$\rho(\alpha_1, ..., \alpha_n) \qquad \text{or} \qquad (\alpha_1 = \alpha_2)$$

where ρ is a relation symbol of arity n, and each α_i is either a constant or a variable.

A first-order logic formula is either an atomic formula or a (finite) expression of one of the forms:

$$\neg\alpha, \qquad (\alpha \; \kappa \; \beta), \qquad \forall x[\alpha], \qquad \exists x[\alpha]$$

where α and β are first-order formulae and κ is any of the binary connectives ($\wedge$, $\vee$, $\rightarrow$ or $\leftrightarrow$).

Restrictions on Quantification

Although the standard semantics for first-order logic will assign a meaning to any formula fitting the stipulation on the previous slide, sensible formulae satisfy some further conditions:

- For every quantification $\forall\xi[\alpha]$ or $\exists\xi[\alpha]$ there is at least one further occurrence of the variable ξ in α.
- No quantification occurs within the scope of another quantification using the same variable.
- Every variable occurs within the scope of a quantification using that variable.

(The *scope* of a symbol σ in formula ϕ is the smallest sub-expression of ϕ which contains σ and is a first-order formula.)

Simple Examples using Relations and Quantifiers

Tom talks to Mary	$\mathsf{TalksTo}(\mathsf{tom}, \mathsf{mary})$
Tom talks to himself	$\mathsf{TalksTo}(\mathsf{tom}, \mathsf{tom})$
Tom talks to everyone	$\forall x[\mathsf{TalksTo}(\mathsf{tom}, x)]$
Everyone talks to tom	$\forall x[\mathsf{TalksTo}(x, \mathsf{tom})]$
Tom talks to no one	$\neg\exists x[\mathsf{TalksTo}(\mathsf{tom}, x)]$
Everyone talks to themself	$\forall x[\mathsf{TalksTo}(x, x)]$
Only Tom talks to himself	$\forall x[\mathsf{TalksTo}(x, x) \leftrightarrow (x = \mathsf{tom})]$

Typical Forms of Quantification

All frogs are green:

$$\forall x[F(x) \to G(x)]$$

Some frogs are poisonous:

$$\exists x[F(x) \wedge P(x)]$$

No frogs are silver:

$$\neg\exists x[F(x) \wedge S(x)]$$

Representing Numbers

In the standard predicate logic, we only have two types of quantifier:

$$\forall x[\phi(x)] \quad \text{and} \quad \exists x[\phi(x)]$$

How can we represent a statement such as 'I saw two birds' ?

What about

$$\exists x\exists y[\mathsf{Saw}(\mathsf{i}, x) \wedge \mathsf{Saw}(\mathsf{i}, y)] \qquad ?$$

This doesn't work. Why?

At Least n

For any natural number n we can specify that there are at least n things satisfying a given condition.

John owns at least two dogs:

$$\exists x\exists y[\mathsf{Dog}(x) \wedge \mathsf{Dog}(y) \wedge \neg(x = y) \wedge \mathsf{Owns}(\mathsf{john}, x) \wedge \mathsf{Owns}(\mathsf{john}, y)]$$

John owns at least three dogs:

$$\exists x\exists y\exists z[\mathsf{Dog}(x) \wedge \mathsf{Dog}(y) \wedge \mathsf{Dog}(z) \wedge \neg(x = y) \wedge \neg(x = z) \wedge \neg(y = z) \wedge \mathsf{Owns}(\mathsf{john}, x) \wedge \mathsf{Owns}(\mathsf{john}, y) \wedge \mathsf{Owns}(\mathsf{john}, z)]$$

At Most n

Every student owns at most one computer:

$$\forall x[\mathsf{Student}(x) \to \neg\exists y\exists z[\mathsf{Comp}(y) \wedge \mathsf{Comp}(z) \wedge \neg(y = z) \wedge \mathsf{Owns}(x, y) \wedge \mathsf{Owns}(x, z)]\ ]$$

or equivalently

$$\forall x\forall y\forall z[(\ \mathsf{Student}(x) \wedge \mathsf{Comp}(y) \wedge \mathsf{Comp}(z) \wedge \wedge\ \mathsf{Owns}(x, y) \wedge \mathsf{Owns}(x, z)\ ) \ \to\ (y = z)]$$

Exactly n

To state that a property holds for exactly n objects, we need to assert that it holds for at least n objects, but deny that it holds for at least $n + 1$ objects:

'A triangle has (exactly) 3 sides':

$$\begin{aligned}
&\forall t[\mathsf{Triangle}(t) \to \\
&\quad (\exists x\exists y\exists z[\mathsf{SideOf}(x, t) \wedge \mathsf{SideOf}(y, t) \wedge \mathsf{SideOf}(z, t) \wedge \\
&\qquad \neg(x = y) \wedge \neg(y = z) \wedge \neg(x = z)] \\
&\quad \wedge \\
&\neg\exists x\exists y\exists z\exists w[\mathsf{SideOf}(x,t) \wedge \mathsf{SideOf}(y,t) \wedge \mathsf{SideOf}(z,t) \wedge \mathsf{SideOf}(w,t) \\
&\qquad \wedge \neg(x = y) \wedge \neg(y = z) \wedge \neg(x = z)] \\
&\quad) \\
&]
\end{aligned}$$

Knowledge Representation

Lecture KRR-6

Classical Logic IV:
Semantics for Predicate Logic

The Domain of Individuals

Whereas a model for propositional logic assigns truth values directly to propositional variables, in predicate logic the truth of a proposition depends on the meaning of its constituent predicate and argument(s).
The arguments of a predicate may be either *constant names* (a, b, . . .) or *variables* (u, v, . . ., z).

To formalise the meaning of these argument symbols each predicate logic model is associated with a set of entities that is usually called the *domain of individuals* or the *domain of quantification*. (Note: Individuals may be anything — either animate or inanimate, physical or abstract.)

Each constant name *denotes* an element of the domain of individuals and variables are said to *range over* this domain.

Semantics for Property Predication

Before proceeding to a more formal treatment of predicate, I briefly describe the semantics of property predication in a semi-formal way.

A property is formalised as a 1-place predicate — i.e. a predicate applied to one argument.

For instance Happy(jane) ascribes the property denoted by Happy to the individual denoted by jane.

To give the conditions under which this assertion is true, we specify that Happy denotes the *set* of all those individuals in the domain that are happy.

Then Happy(jane) is true just in case the individual denoted by jane is a member of the set of individuals denoted by Happy.

Predicate Logic Model Structures

A predicate logic model is a tuple

$$\mathcal{M} = \langle \mathcal{D}, \delta \rangle ,$$

where:

- $\mathcal{D}$ is a non-empty set (the domain of individuals) —
  i.e. $\mathcal{D} = \{i_1, i_2, \ldots\}$, where each i_n represents some entity.
- δ is an assignment function, which gives a value to each constant name and to each predicate symbol.

The Assignment Function δ

The kind of value given to a symbol σ by the assignment function δ depends on the type of σ:

- If σ is a constant name then $\delta(\sigma)$ is simply an element of $\mathcal{D}$.
  (E.g. $\delta(\text{john})$ denotes an individual called 'John'.)
- If σ is a property, then $\delta(\sigma)$ denotes a *subset* of the elements of $\mathcal{D}$.
  This is the subset of all those elements that possess the property σ. (E.g. $\delta(\text{Red})$ would denote the set of all red things in the domain.)
- *continued on next slide for case where σ is a relation symbol.*

The Assignment Function for Relations

- If σ is a binary relation, then $\delta(\sigma)$ denotes a *set of pairs* of elements of $\mathcal{D}$.
  For example we might have
  $$\delta(R) = \{\langle i_1, i_2\rangle, \langle i_3, i_1\rangle, \langle i_7, i_2\rangle, \ldots\}$$
  The value $\delta(R)$ denotes the set of all pairs of individuals that are related by the relation R.
  (Note that we may have $\langle i_m, i_n\rangle \in \delta(R)$ but $\langle i_n, i_m\rangle \notin \delta(R)$ — e.g. John loves Mary but Mary does not love John.)
- More generally, if σ is an n-ary relation, then $\delta(\sigma)$ denotes a *set of n-tuples* of elements of $\mathcal{D}$.
  (E.g. $\delta(\text{Between})$ might denote the set of all triples of points, $\langle p_x, p_y, p_z\rangle$, such that p_y lies between p_x and p_z.)

The Semantics of Predication

We have seen how the denotation function δ assigns a value to each individual constant and each relation symbol in a predicate logic language.

The purpose of this is to define the conditions under which a predicative proposition is true.
Specifically, a predication of the form $\rho(\alpha_1, \dots \alpha_n)$ is true according to δ if and only if

$$\langle \delta(\sigma_1), \dots \delta(\sigma_n) \rangle \in \delta(\rho)$$

For instance, $\mathsf{Loves}(\mathsf{john}, \mathsf{mary})$ is true iff the pair $\langle \delta(\mathsf{john}), \delta(\mathsf{mary}) \rangle$ (the pair of individuals denoted by the two names) is an element of $\delta(\mathsf{Loves})$ (the set of all pairs, $\langle i_m, i_n \rangle$, such that i_m loves i_n).

Variable Assignments and Augmented Models

In order to specify the truth conditions of quantified formulae we will have to interpret variables in terms of their possible values.

Given a model $\mathcal{M} = \langle \mathcal{D}, \delta \rangle$, Let V be a function from variable symbols to entities in the domain $\mathcal{D}$.

I will call a pair $\langle \mathcal{M}, V \rangle$ an *augmented model*, where V is a variable assignment over the domain of $\mathcal{M}$.

If an assignment V' gives the same values as V to all variables except possibly to the variable x, I write this as:

$$V' \approx_{(x)} V\ .$$

This notation will be used in specifying the semantics of quantification.

Truth and Denotation in Augmented Models

We will use augmented models to specify the truth conditions of predicate logic formulae, by stipulating that ϕ is true in $\mathcal{M}$ if and only if ϕ is true in a corresponding augmented model $\langle \mathcal{M}, V \rangle$.

It will turn out that if a formula is true in *any* augmented model of $\mathcal{M}$, then it is true in *every* augmented model of $\mathcal{M}$. The purpose of the augmented models is to give a denotation for variables.

From an augmented model $\langle \mathcal{M}, V \rangle$, where $\mathcal{M} = \langle \mathcal{D}, \delta \rangle$, we define the function δ_V, which gives a denotation for both constant names and variable symbols. Specifically:

- $\delta_V(\alpha) = \delta(\alpha)$, where α is a constant;
- $\delta_V(\xi) = V(\xi)$, where ξ is a variable.

Semantics for the Universal Quantifier

We are now in a position to specify the conditions under which a universally quantified formula is true in an augmented model:

- $\forall x[\phi(x)]$ is true in $\langle \mathcal{M}, V \rangle$ iff
  $\phi(x)$ is true in every $\langle \mathcal{M}, V' \rangle$, such that $V' \approx_{(x)} V$.

In other words this means that $\forall x[\phi(x)]$ is true in a model just in case the sub-formula $\phi(x)$ is true whatever entity is assigned as the value of variable x, while keeping constant any values already assigned to other variables in ϕ.

We can define existential quantification in terms of universal quantification and negation; but what definition might we give to define its semantics directly?

Semantics of Equality

In predicate logic, it is very common to make use of the special relation of *equality*, '='.

The meaning of '$=$' can be captured by specifying axioms such as

$$\forall x \forall y[((x = y) \wedge \mathsf{P}(x)) \rightarrow \mathsf{P}(y)]$$

of by means of more general inference rules such as,
from $(\alpha = \beta)$ and $\phi(\alpha)$ derive $\phi(\beta)$.

We can also specify the truth conditions of equality formulae using our augmented model structures:

- $(\alpha = \beta)$ is true in $\langle M, V \rangle$, where $\mathcal{M} = \langle \mathcal{D}, \delta \rangle$,
  iff $\delta_V(\alpha)$ is the same entity as $\delta_V(\beta)$.

Full Semantics of Predicate Logic

- $\rho(\alpha_1, \dots \alpha_n)$ is true in $\langle \mathcal{M}, V \rangle$, where $\mathcal{M} = \langle \mathcal{D}, \delta \rangle$,
  iff $\langle \delta_V(\sigma_1), \dots \delta_V(\sigma_n) \rangle \in \delta(\rho)$.
- $(\alpha = \beta)$ is true in $\langle M, V \rangle$, where $\mathcal{M} = \langle \mathcal{D}, \delta \rangle$, iff $\delta_V(\alpha) = \delta_V(\beta)$.
- $\neg\phi$ is true in $\langle \mathcal{M}, V \rangle$ iff ϕ is *not* true in $\langle \mathcal{M}, V \rangle$
- $(\phi \wedge \psi)$ is true in $\langle \mathcal{M}, V \rangle$ iff both ϕ and ψ are true in $\langle \mathcal{M}, V \rangle$
- $(\phi \vee \psi)$ is true in $\langle \mathcal{M}, V \rangle$ iff either ϕ or ψ is true in $\langle \mathcal{M}, V \rangle$
- $\forall x[\phi(x)]$ is true in $\langle \mathcal{M}, V \rangle$ iff
  $\phi(x)$ is true in every $\langle \mathcal{M}, V' \rangle$, such that $V' \approx_{(x)} V$.

Knowledge Representation

Lecture KRR-7

The Winograd Schema Challenge

The Winograd Schema Challenge

This challenge has been proposed as a more well-defined alternative to the famous Turing Test. It brings many problems of AI and KRR more sharply into focus than does the rather open-ended Turing Test.

- Corpus of Winograd Schema problems — by Ernie Davis
  `http://www.cs.nyu.edu/davise/papers/WS.html`
- Paper on Winograd Schema problems by Hector Levesque
  `http://commonsensereasoning.org/2011/papers/Levesque.pdf`

Nature of the Challenge

A Winograd schema is a pair of sentences that differ in only one or two words and that contain an ambiguity that is resolved in opposite ways in the two sentences and requires the use of world knowledge and reasoning for its resolution.

The schema takes its name from a well-known example by Terry Winograd (1972)

The city councilmen refused the demonstrators a permit because *they* [feared/advocated] violence.

If the word is “feared”, then “they” presumably refers to the city council; if it is “advocated” then “they” presumably refers to the demonstrators.

Requirements for a good schema

In his paper, “The Winograd Schema Challenge” Hector Levesque (2011) proposes to assemble a corpus of such Winograd schemas that are

- easily disambiguated by the human reader (ideally, the reader does not even notice there is an ambiguity);
- not solvable by simple techniques such as selectional restrictions
- Google-proof; ie no *statistical* test over text corpora that will reliably disambiguate.

The corpus would then be presented as a challenge for AI programs, along the lines of the Turing test. The strengths of the challenge are that it is more clear-cut.

More Winograd Schemas

- The trophy would not fit into the brown suitcase because it was too [small/large]. What was too [small/large]?
- Joan made sure to thank Susan for all the help she had [given/received]. Who had [given/received] help?
- The large ball crashed right through the table because it was made of [steel/styrofoam] What was made of [steel/styrofoam]?

vi

Knowledge Representation

Lecture KRR-8

Representing Time and Change

Lecture Overview

This lecture has the following goals:

- to demonstrate the importance of temporal information in knowledge representation.
- to introduce two basic logical formalisms for describing time (1st-order temporal logic and Tense Logic).
- to present two AI formalisms for representing actions and change (STRIPS and Situation Calculus).
- to explain *Frame Problem* and some possible solutions.

Classical Propositions are Eternal

A classical proposition is either true or false.

So it cannot be true sometimes and false at other times.

Hence a *contingent* statement such as '*Tom is at the University*' does not really express a classical proposition.

Its truth depends on when the statement is made.

A corresponding classical proposition would be something like:
Tom was/is/will be at the University at 11:22am 8/2/2002.
This statement, if true, is eternally true.

Building Time into 1st-order Logic

We can explicitly add time references to 1st-order formulae. For example

$$\text{Happy}(\text{John}, t)$$

could mean 'John is happy at time t'.

In this representation each predicate is given an extra argument place specifying the time at which it is true.

Time as an Ordering of Time Points

To talk about the ordering of time points we introduce the special relation $\leq$. Being a (linear) order it satisfies the following axioms.

1. $\forall t_1 \forall t_2 \forall t_3 [(t_1 \leq t_2 \wedge t_2 \leq t_3) \rightarrow t_1 \leq t_3]$, (transitivity)
2. $\forall t_1 \forall t_2 [t_1 \leq t_2 \ \vee \ t_2 \leq t_1]$, (linearity)
3. $\forall t_1 \forall t_2 [(t_1 \leq t_2 \wedge t_2 \leq t_1) \leftrightarrow t_1 = t_2]$, (anti-symmetry)

We can define a *strict* ordering relation by:

$$t_1 < t_2 \ \equiv_{def} \ t_1 \leq t_2 \wedge \neg(t_1 = t_2)$$

Some Further Possible Axioms

Time will continue infinitely in the future:

$$\forall t \exists t' [t < t']$$

What does the following axiom say?

$$\forall t_1 \forall t_2 [(t_1 < t_2) \rightarrow \exists t_3 [(t_1 < t_3) \wedge (t_3 < t_2)]$$

. . . .

Representing Temporal Ordering

Sue is happy but will be sad:

$$\text{Happy}(\text{Sue}, 0) \wedge \exists t[(0 < t) \wedge \text{Sad}(\text{Sue}, t)]$$

Here I use 0 to stand for the present time.

We can describe more complex temporal constraints of a *causal* nature.
E.g. 'When the sun comes out I am happy until it rains':

$$\forall t[S(t) \rightarrow \forall u[(t \leq u \wedge \neg\exists r[(t \leq r) \wedge (r \leq u) \wedge R(r)]) \rightarrow H(u)]]$$

Another Way of Adding Time

Rather than adding time to each predicate, several AI researchers have found it more convenient to use a special type of relation between propositions and time points:

$$\text{Holds-At}(\text{Happy}(\text{John}),\ t)$$

Can use this to define temporal relations in a more general way. E.g.:

$$\forall t[\text{Holds-At}(\phi, t) \rightarrow \exists t'[t \leq t' \wedge \text{Holds-At}(\psi, t')]$$

This captures a possible specification of the relation 'ϕ *causes* ψ'.

Axioms for Holds-At

1. $(\text{Holds-At}(\phi, t) \wedge \text{Holds-At}(\phi \rightarrow \psi, t)) \rightarrow \text{Holds-At}(\psi, t)$
2. $\neg\text{Holds-At}(\phi \wedge \neg\phi, t)$
3. $\text{Holds-At}(\phi, t) \vee \text{Holds-At}(\neg\phi, t)$
4. $\text{Holds-At}(\phi, t) \leftrightarrow \text{Holds-At}(\text{Holds-At}(\phi, t), t')$
5. $t \leq t' \leftrightarrow \text{Holds-At}((t \leq t'), t'')$
6. $\forall t[\text{Holds-At}(\phi, t')] \rightarrow \text{Holds-At}(\forall t[\phi], t')$

Tense Logic

Rather than quantifying over time points, it may be simpler to treat time in terms of *tense*.

$\mathbf{P}\phi$ means that ϕ was true at some time in the past. $\mathbf{F}\phi$ means that ϕ will be true at some time in the future.

If Jane has arrived I will visit her:

$$\mathbf{P}A(j) \rightarrow \mathbf{F}V(j)$$

Axioms for Tense Operators

The tense operators obey certain axioms. For example:

1. $\mathbf{F}\phi \rightarrow \neg\mathbf{P}\neg\mathbf{F}\phi$
2. $\mathbf{P}\phi \rightarrow \neg\mathbf{F}\neg\mathbf{P}\phi$
3. $\mathbf{PP}\phi \rightarrow \mathbf{P}\phi$
4. $\mathbf{FF}\phi \rightarrow \mathbf{F}\phi$

Can you think of any more?

Prior's Tense Logic

It is convenient to define:

ϕ *has always been true*	ϕ *will always be true*
$\mathbf{H}\phi \equiv_{def} \neg\mathbf{P}\neg\phi$	$\mathbf{G}\phi \equiv_{def} \neg\mathbf{F}\neg\phi$

We now specify the following axioms:

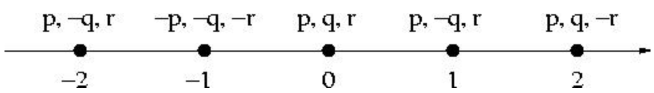
1)	$(\mathbf{H}\phi \wedge \mathbf{H}(\phi \rightarrow \psi)) \rightarrow \mathbf{H}\psi$	2)	$(\mathbf{G}\phi \wedge \mathbf{G}(\phi \rightarrow \psi)) \rightarrow \mathbf{G}\psi$
3)	$\phi \rightarrow \mathbf{HF}\phi$	4)	$\phi \rightarrow \mathbf{GP}\phi$
5)	$\mathbf{P}\phi \rightarrow \mathbf{GP}\phi$	6)	$\mathbf{F}\phi \rightarrow \mathbf{HF}\phi$
7)	$\mathbf{P}\phi \rightarrow \mathbf{H}(\mathbf{F}\phi \vee \phi \vee \mathbf{P}\phi)$	8)	$\mathbf{F}\phi \rightarrow \mathbf{G}(\mathbf{F}\phi \vee \phi \vee \mathbf{P}\phi)$
9)	$\mathbf{P}(\phi \vee \neg\phi)$	10)	$\mathbf{F}(\phi \vee \neg\phi)$

Together with any sufficient axiom set for classical propositional logic.

Models for Tense Logics

A tense logic model is given by a set $\mathcal{M} = \{\ldots, M_i, \ldots\}$ of atemporal classical models, whose indices are ordered by a relation $\prec$.

The models can be pictured as corresponding to different moments along the time line:



Validity in Tense Logic Models

Truth values of (atemporal) classical formulae are determined by each model as usual. A classical formula is true at index point i iff it is true according to M_i.

Tensed formulae are interpreted by:

- $\mathbf{F}\phi$ is true at i iff ϕ is true at some j such that $i \prec j$.
- $\mathbf{P}\phi$ is true at i iff ϕ is true at some j such that $j \prec i$.

A tense logic formula is *valid* iff it is true at every index point in every tense logic model.

Reasoning with Tense Logic

Reasoning directly with tense logic is extremely difficult. We need to combine classical propositional reasoning with substitution in the axioms.

Exercise: try to prove that $\mathbf{PP}p \rightarrow \mathbf{P}p$ from Prior's axioms. I couldn't !

But *Model Building* techniques can be quite efficient.

A model is an ordered set of time points, each associated with a set of formulae.

A proof algorithm can exhaustively search for a model satisfying any given formula.

STRIPS

The STanford Research Institute Planning System is a relatively simple algorithm for reasoning about actions and formulating plans.
STRIPS models a state of the world by a set of (atomic) facts. Actions are modelled as rules for adding and deleting facts.

Specifically each action definition consists of:

	for example
Action Description:	`move(x, loc1, loc2)`
Preconditions:	`at(x, loc1), movable(x), free(loc2)`
Delete List:	`at(x, loc1), free(loc2)`
Add List:	`at(x, loc2), free(loc1)`

Goal-Directed Planning

The STRIPS system enables a relatively straightforward implementation of goal-directed planning.

To find a plan to achieve a goal `G` we can use an algorithm of the following form:

1. If `G` is already in the set of world facts we have succeeded.
2. Otherwise look for an action definition
   $(\alpha,\ [\pi_1, \ldots, \pi_l],\ [\delta_1, \ldots, \delta_m],\ [\gamma_1, \ldots,\ \mathtt{G},\ \ldots, \gamma_n])$
   with `G` in its add list.
3. Then successively set each precondition π_i as a new sub-goal and repeat this procedure.

More complex search strategy is needed for good performance.

Limitations of STRIPS

STRIPS works well in cases where the effects of actions can be captured by simple adding and deleting of facts. However, for general types of action that can be applied in a variety of circumstances, the effects are often highly dependent on context.

Even with the simple action `move(x,loc1,loc2)` the changes in facts involving `x` will depend on what other objects are near to `loc1` and `loc2`.

In general the interdependencies of even simple relationships. Are highly complex. Consider the ways in which a relation `visible-from(x,y)` can change — e.g. when crates are moved around in a warehouse.

Situation Calculus

Situation Calculus is a 1st-order language for representing dynamically changing worlds.

Properties of a *state* of the world are represented by:

- $holds(\phi, s)$ meaning that 'proposition' ϕ holds in state s.

In the terminology of Sit Calc ϕ is called a fluent.

Actions in Sit Calc

In Sit Calc all changes are the result of actions:

- $result(\alpha, s)$ denotes the state resulting from doing action α when in state s.

We can write formulae such as:

$$holds(\text{Light-Off}, s) \rightarrow holds(\text{Light-On}, result(\textbf{switch}, s))$$

Effect Axioms

Effect axioms specify fluents that must hold in the state resulting from an action of some given type.

Simple effect axioms can be written in the form:

$$holds(\phi, result(\alpha, s)) \leftarrow poss(\alpha, s)$$

The reverse arrow is used so that the most important part is at the beginning. It also corresponds to form used in Prolog implementations.

Here $poss$ is an auxiliary predicate that is often used to separate the preconditions of an action from the rest of the formula.

$$holds(\text{has}(y, i), result(\mathbf{give}(x, i, y), s)) \leftarrow poss(\mathbf{give}(x, i, y), s)$$

Precondition Axioms

Preconditions tell us what fluents must hold in a situation for it to be possible to carry out a given type of action in that situation.

If we are using the $poss$ predicate, a simple precondition takes the form:

$$poss(\alpha, s) \leftarrow holds(\phi, s)$$

Example:

$$poss(\mathbf{give}(x, i, y), s) \leftarrow holds(\text{has}(x, i), s)$$

More Examples

$$poss(\mathbf{mend}(x, i), s) \leftarrow holds((\text{has}(x, i) \wedge \text{broken}(i) \wedge \text{has}(x, \text{glue})), s)$$

$$poss(\mathbf{steal}(x, i, y), s) \leftarrow holds((\text{has}(y, i) \wedge \text{asleep}(y) \wedge \text{stealthy}(x)), s)$$

Domain Axioms

As well as axioms describing the transition from one state to another actions and their effects, a Situation Calculus theory will often include *domain axioms* specifying conditions that must hold in every possible situation.

As well as fluents, a Sit Calc theory may utilise static predicates expressing properties that do not change.

$\text{ConnectedByDoor}(\text{kitchen}, \text{dining\_room}, \text{door1})$
$\forall r_1 r_2 d[\text{ConnectedByDoor}(r_1, r_2, d) \rightarrow \text{ConnectedByDoor}(r_2, r_1, d)]$

Other domain axioms may express relationships between fluents that must hold in every situation.

$$\forall s \forall x[\neg(holds(\text{happy}(x), s) \wedge holds(\text{sad}(x), s)]$$

Frame Axioms

Frame axioms tell us what *fluents* do not change when an action takes place.

When you're dead you stay dead:

$$holds(\textsf{dead}(x), result(\alpha, s)) \leftarrow holds(\textsf{dead}(x), s)$$

Giving something won't mend it:

$$holds(\textsf{broken}(i), result(\mathbf{give}(x, i, y), s)) \leftarrow holds(\textsf{broken}(i), s)$$

More generally, we might specify that no action apart from $\mathbf{mend}$ can mend something:

$$holds(\textsf{broken}(i), result(\alpha, s)) \leftarrow holds(\textsf{broken}(i), s) \wedge \neg\exists x[\alpha = \mathbf{mend}(x, i)]$$

The Frame Problem

Intuitively it would seem that, if we specify all the effects of an action, we should be able to infer what it doesn't affect.

We would like to have a general way of automatically deriving reasonable frame conditions.

The *frame problem* is that no completely general way of doing this has been found.

Solving the Frame Problem

The AI literature contains numerous suggestions for solving the frame problem.

None commands universal acceptance.

There are two basic approaches:

- Syntactic derivation of frame axioms from effect axioms.
- Use of *Non-Monotonic* reasoning techniques.

Ramifications

Events and Intervals

Tense logic and logics with explicit time variables represent change in terms of what is true along a series of time *points*. They have no way of saying that some event or process happens over some *interval* of time.

A conceptualisation of time in terms of intervals and events was proposed by James Allen (and also Pat Hayes) in the early 80's.

The formalism contains variables standing for temporal intervals and terms denoting types of event.

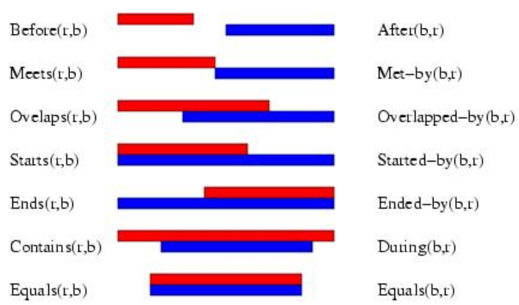
We can use basic expressions of the form:

$$\textsf{Occurs}(\mathbf{action}, i)$$

saying that $\mathbf{action}$ occurs over time interval i.

Allen's Interval Relations

Allen also identified 13 qualitatively different relations that can hold between termporal intervals:



Ordering Events

By combining the occurs relation with the interval relations we can describe the ordering of events:

Occurs($\mathbf{get\_dressed}, i$)
Occurs($\mathbf{travel\_to\_work}, j$)
Occurs($\mathbf{read\_newspaper}, k$)
Before(i, j)
During(k, j)

What can we infer about the temporal relation between $\mathbf{get\_dressed}$ and $\mathbf{read\_newspaper}$?

ii

iii

iv

v

vi

Knowledge Representation

Lecture KRR-9

Prolog

Reasons to Learn a Bit of Prolog

- The *Logic Programming* paradigm is radically different from the traditional imperative style; so knowledge of Prolog helps develop a broad appreciation of programming techniques.
- Although not usually employed as a general purpose programming language, Prolog is well-suited for certain tasks and is used in many research applications.
- Prolog has given rise to the paradigm of *Constraint Programming* which is used commercially in scheduling and optimisation problems.

Pitfalls in Learning Prolog

One reason that Prolog is not more widely used is that a beginner can often encounter some serious difficulties.

Trying to crowbar an imperative algorithm into Prolog syntax will generally result in complex, ugly and often incorrect code. A good Prolog solution must be formulated from the beginning in the logic programming idiom. Forget loops and assignments and think in terms of Prolog concepts.

Another difficulty is in getting to grips the execution sequence of Prolog programs. This is subtle because Prolog is written declaratively. Nevertheless, its search for solutions does proceed in a determinate fashion and unless code is carefully ordered it may search forever, even when a solution exists.

Prolog Structures

You should be aware that coding for many kinds of application is facilitated by representing data in a structured way. Prolog (like Lisp) provides generic syntax for structuring data that can be applied to all manner of particular requirements.

A complex term takes the following form:

```
operator(arg1, ..., argN)
```

Where `arg1, ..., argN` may also be complex terms.

Although such a term looks like a function, it is *not evaluated* by applying the function to the arguments. Instead Prolog tries to find values for which it is true by matching it to the facts and rules given in the code.

Pattern Matching and Equality

When evaluating a query containing one or more variables, Prolog tries to match the query to a stored (or derivable) fact, in which variables may be replaced by particular instances.

In fact the matching always goes both ways. If we have the code:

```
likes( X, X ).
```

Then the query `?- likes( john, john )` will give `yes`, even though there are no facts given about `john`, because the query matches the general fact — which says that everyone likes themself.

Matching Complex Terms

Prolog will also find matches of complex terms as long as there is a instantiation of variables that makes the terms equivalent:

```
loves( brother(X), daughter(tom) ).

?- loves( brother( susan ), Y ).
X = susan,
Y = daughter(tom)
```

We can make Prolog perform a match using the equality operator:

```
?-  f( g(X), this, X )  =  f( Z, Y, that ).
X = that,
Y = this,
Z = g(that)
```

Note that no evaluation of `f` and `g` takes place.

Coding by Matching

A surprising amount of functionality can be achieved simply by pattern matching.

Consider the following code:

```
vertical( line(point(X,Y), point(X,Z)) ).
horizontal( line(point(X,Y), point(Z,Y)) ).
```

Here we are assuming a representation of line objects as pairs of point objects, which in turn are pairs of coordinates.

Given just these simple facts, we can ask queries such as:

```
?- vertical( line(point(5,17), point(5,23)) ).
```


The List Constructor Operator

The basic syntax that is used either to construct or to break up lists in Prolog is the head-tail structure:

[*Head* | *Tail*]

Here *Head* is any term. It could be an atom, a variable or some complex structure.

Tail is either a variable or any kind of list structure.

The structure [*Head* | *Tail*] denotes the list formed by adding *Head* at the front of the list *Tail*.

Thus `[a | [b,c,d]]` denotes the list `[a,b,c,d]`.

Matching Lists

Consider the following query:

```
?-   [ H | T ] = [a, b, c, d, e].
```

This will return:

```
Head = a,
Tail = [b,c,d,e]
```

Note that this is identical in meaning to:

```
?-    [a, b, c, d, e] = [ H | T ].
```


Recursion Over Lists

The following example is of utmost significance in illustrating the nature of Prolog. It combines, pattern matching, lists and recursive definition. Learn this:

```
in_list( X, [X | _] ).
in_list( X, [_ | Rest] ) :- in_list( X, Rest ).
```

Given this definition, `in_list(`*elt*`,` *list*`)` will be true, just in case *elt* is a member of *list*.

(In fact this same functionality is already provided by the inbuilt `member` predicate.)

Check if Arrays Share a Value in C

```
#include <stdio.h>

int main() {
    int arrayA [4] = {1,2,3,4};
    int arrayB [3] = {3,6,9};
    if ( arrays_share_value( arrayA, 4, arrayB, 3 ) )
          printf("YES\n");
     else printf("NO\n");
}
int arrays_share_value( int A1[], int len1,  int A2[], int len2 ) {
    int i, j;
    for (i = 0; i < len1; i++ ) {
        for (j = 0; j < len2; j++) {
               if (A1[i] == A2[j]) return 1;
            }
        }
    return 0;
}
```


Check if Lists Share an Element in Prolog

```
lists_share_element( L1, L2 ) :-
        member( X, L1 ),
        member( X, L2 ).

?- lists_share_element( [1,2,3,4], [3,6,9] ).
yes
```

More Useful Prolog Operators and Built-Ins

We shall now briefly look at some other useful Prolog constructs using the following operators and built-in predicates:

- math evaluation: `is`
- negation: `\+`
- disjunction: `;`
- `setof`
- cut: `!`

Math Evaluation with 'is'

Though possible, it would be rather tedious and very inefficient to code basic mathematical operations in term of Prolog facts and rules (e.g. `add(1,1,2).`, `add(1,2,3).` etc).

However, the '`is`' enables one to evaluate a math expression and bind the value obtained to a variable.

For example after executing the code line

```
X is sqrt( 57 + log(10) )
```

Prolog will bind `X` to the appropriate decimal number:

```
X = 7.700817170469251
```


The Negation Operator, '\+'

It is often useful to check that a particular goal expression does not succeed.

This is done with the '\+' operator.

E.g.

```
\+loves(john, mary)

\+loves(john, X )
```

By using brackets one can check wether two or more predicates cannot be simultaneously satisfied:

```
\+( member( Z, L1), member(Z, L2 ) )
```


Forming Disjunctions with ';'

Sometimes we may want to test whether either of two goals is satisfied. We can do this with an expression such as:

```
( handsome(X) ; rich(X) )
```

This will be true if there is a value of `X`, such that *either* the `handsome` or the `rich` predicate is true for this value.

Thus, we could define:

```
attractive( X ) :- (handsome(X) ; rich(X) ).
```

This is actually equivalent to the pair of rules:

```
attractive( X ) :- handsome(X).
attractive( X ) :- rich(X).
```


Combining Operators

In general, we can combine several operators to form a complex goal which is a (truth functional) combination of several simpler goals.

E.g.:

```
eligible( X ) :- handsome( X ),
                 (single(X) ; rich(X)),
                 \+ cheesy_grin( X ).
```


The `setof` Predicate

It is often very useful to be able to find the set of all possible values that satisfy a given predicate.

This can be done with the special built-in `setof` predicate, which is used in the following way:

```
setof( X, some_predicate(X), L )
```

This is true if `L` is a list whose members are all those values of `X`, which satisfy the predicate `some_predicate(X)`.

E.g.:

```
eligible_shortlist( L ) :-
                  setof( X, eligible(X), L ).
```

The *Cut* Operator, '!'

The so-called *cut* operator is used to control the flow of execution, by preventing Prolog backtrack past the cut to look for alternative solutions.
Consider the following mini-program:

```
red(a).         black(b).
color(P,red)     :- red(P), !.
color(P,black)   :- black(P), !.
color(P,unknown).
```

Consider what happens if we give the following queries:

```
?- color(a, Col).
?- color(c, Col).
```

What would be the difference if the cuts were removed?

Data Record Processing Example

Here is a typical example of how some data might be stored in the form of Prolog facts:

```
%%%       ID    Surname      First Name   User Name     Degree
student( 101, 'SMITH',        'John',     comp2010js, 'Computing' ).
student( 102, 'SMITH',        'Sarah',    comp2010ss, 'Computing' ).
student( 103, 'JONES',        'Jack',     comp2010jj, 'Computing' ).
student( 104, 'DE-MORGAN' 'Augustus', log2010adm, 'Logic' ).
student( 105, 'RAMCHUNDRA', 'Lal',     log2010lr,  'Logic' ).

coursework( 1, comp2010js,  45 ).
coursework( 1, comp2010ss,  66 ).
coursework( 1, comp2010jj,  35 ).
coursework( 1, log2010adm,  99 ).
coursework( 1, log2010lr,   87 ).
```


Deriving Further Info from the Data

Given the data format used on the previous slide, the following code concisely defines how to derive the top mark for a given coursework:

```
top_mark( CW, [First, Sur] ) :-
         coursework( CW, User, Mark1 ),
         \+( ( coursework( CW, _, Mark2 ),
               Mark2 > Mark1
             )
           ),
         student( _, Sur, First, User, _ ).
```

Note: the extra layer of bracketing in `\+(( ... ))` is required in order to compound two predicates to form a single argument for the \+ operator.

More Data Extraction Rules

Here are some further useful rules for extracting information from the student and coursework data:

```
student_user( U ) :- student( _, _, _, U, _).

user_pass_cw( U, CW ) :-
          coursework( CW, U, Mark ),
          Mark >= 40.

user_name( U, [S,F] ) :- student( _, S, F, U, _).
```


Example Use of `setof`

The `setof` operator enables us to straightforwardly derive the whole set of elements satisfying a given condition:

```
pass_list( CW, PassList ) :-
       setof( Name, U^(user_pass_cw(U,CW),
                       user_name(U,Name)), PassList
            ).
```

Note: the construct `U^( ... )` is a special operator that is used within setof commands to tell prolog that the value of variable `U` can be different for each member of the set.

Conclusion

The nature of Prolog programming is very different from other languages.

In order to program efficiently you need to understand typical Prolog code examples and build your programs using similar style.

Trying to put imperative ideas into a decarative form can lead to overly complex and error prone code.

Although Prolog code is very much declarative in nature, in order for programs to work correctly and efficiently one must also be aware of how code ordering affects the execution sequence.

Knowledge Representation

Lecture KRR-10

Spatial Reasoning

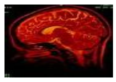
Importance of Spatial Information

Spatial Information is crucial to many important types of software systems. For example:

- Geographical Information Systems (GIS)

- Robotic Control

- Medical Imaging



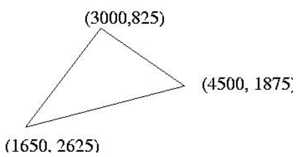
- Virtual Worlds and Video Games

Quantitative Approaches

Most existing computer programs that handle spatial information employ a *quantitative* representation, based on numerical coordinates.

A polygon is represented by a list of the coordinates of its vertices.

For example, a triangle in 2D space is represented by six numbers — two for each of its three corners.



Qualitative Representations

An approach to spatial reasoning, which is becoming increasingly popular in AI (and has been studied for some time at Leeds) is to use *qualitative* representations.

Qualitative representation use high level concepts to describe spatial properties and configurations.

E.g. P(x, y) can mean that region x is a *part* of region y.

Qualitative spatial theories may be formulated in a standard logical language, such as 1st-order logic.

However the use of special purpose reasoning methods, such as *compositional reasoning* is often better than using general 1st-order reasoning methods.

Logical Primitives for Geometry

The (Euclidean) geometry of points can be axiomatised in terms of the single primitive of equidistance which holds between two pairs of points.

EQD(x, y, z, w) holds just in case dist(x, y) = dist(z, w).

Equidistance satisfies the following axioms:

$\forall xy[\mathsf{EQD}(x, y, y, x)]$	reflexivity
$\forall xyz[\mathsf{EQD}(x, y, z, z) \rightarrow (x = y)]$	identity
$\forall xyzuvw[(\mathsf{EQD}(x, y, z, u) \wedge \mathsf{EQD}(x, y, v, w))$ $\rightarrow \mathsf{EQD}(z, u, v, w)]$	transitivity

(A *complete* axiomatisation requires quite a few more axioms.)

Points Lines and Regions

In Euclidean geometry (and quantitative representations based upon it) the *point* is taken as a primitive entity.

A line can be defined by a pair of points.

A two or three dimensional region is represented by a *set* of points.

Since sets are computationally unmanageable, one can normally only deal with regions corresponding to a restricted classes of point sets. E.g. polygons, polyhedra, spheres, cylinders, etc.

An irregular region, such as a country, must be represented as a polygon.

Region-Based Representations

A number of qualitative representations have been proposed in which spatial *regions* are taken as primitive entities.

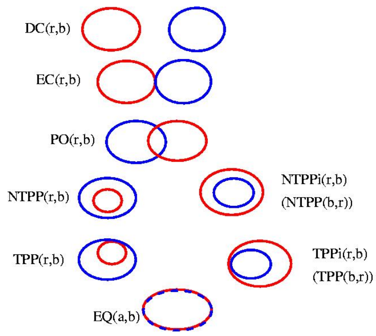
This has several advantages:

- Simple qualitative relations between regions do not need to be analysed as complex relations between their points.
- Natural language expressions often correspond to properties and relations involving regions (rather than points or sets of points).

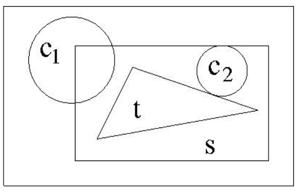
A disadvantage is that logical reasoning using these concepts is often much more complex than numerical computations on point coordinates.

The RCC-8 Relations

The following set of binary topological relations, known as *RCC-8*, has been found to be particularly significant.



Example Topological Description



$$\mathsf{PO}(c_1, s) \wedge \mathsf{DC}(c_1, t) \wedge \mathsf{DC}(c_1, c_2) \wedge$$
$$\mathsf{TPP}(c_2, s) \wedge \mathsf{EC}(c_2, t) \wedge \mathsf{NTPP}(t, s)$$

Connection as a Topological Primitive

A very wide range of topological concepts can be defined in terms of the single primitive relation $\mathsf{C}(x, y)$, which means that region x is *connected* to region y.

This means that x and y either share some common part (they overlap) or they may be only externally connected (the touch).

Connection is *refexive* and *symmetric* — i.e. it satisfies the following axioms:

$$\forall x[\mathsf{C}(x, x)]$$

$$\forall x \forall y[\mathsf{C}(x, y) \rightarrow \mathsf{C}(y, x)]$$

Defining other Relations from Connection

Many other properties and relations can be defined from connection. For example:

$$\mathsf{DC}(x, y) \equiv_{def} \neg\mathsf{C}(x, y)$$

$$\mathsf{P}(x, y) \equiv_{def} \forall z[\mathsf{C}(z, x) \rightarrow \mathsf{C}(z, y)]$$

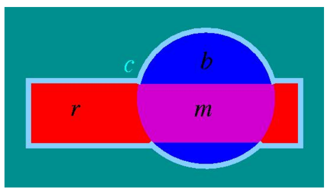
$$\mathsf{DR}(x, y) \equiv_{def} \neg\exists z[\mathsf{P}(z, x) \wedge \mathsf{P}(z, y)]$$

$$\mathsf{EC}(x, y) \equiv_{def} \mathsf{C}(x, y) \wedge \mathsf{DR}(x, y)$$

Defining the Sum of two Regions

One can define a function which gives the sum of two regions:

$$\forall x \forall y \forall z[x = \mathsf{sum}(y, z) \leftrightarrow \forall w[\mathsf{C}(w, x) \leftrightarrow (\mathsf{C}(w, y) \vee \mathsf{C}(w, z)]$$



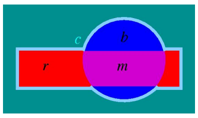
$c = \mathsf{sum}(r, b)$

Intersections

If two regions overlap then there will be a region which is in the intersection of the two. This can be stated by the following axiom:

$$\forall x \forall y[\mathsf{O}(x,y) \leftrightarrow \exists z[\mathsf{INT}(x,y,z)]]$$

So, the following figure satisfies $\mathsf{INT}(r,b,m)$.



The meaning of the INT predicate can be defined by the following equivalence:

$$\mathsf{INT}(x,y,z) \leftrightarrow \forall w[(\mathsf{P}(w,x) \wedge \mathsf{P}(w,y)) \leftrightarrow \mathsf{P}(w,z)]$$

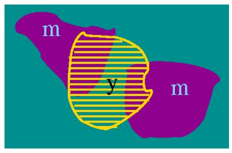
Self-Connectedness

A region that consists of a single connected piece may be called ‘one-piece’, ‘self-connected’ or just ‘connected’ (not to be confused with the binary connection relation).

In terms of sum and binary connection we can define:

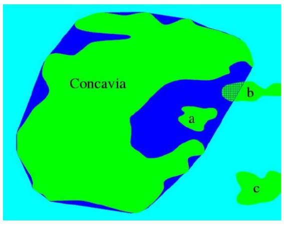
$$\mathsf{SCON}(x) \equiv \forall y \forall z[(x = \mathsf{sum}(y,z)) \rightarrow \mathsf{C}(y,z)]$$

Here we have $\mathsf{SCON}(y) \wedge \neg\mathsf{SCON}(m) \wedge \mathsf{SCON}(\mathsf{sum}(y,m))$.



Convexity

By adding a primitive notion of convexity (e.g. using a *convex hull* function), we can define many properties relating to shape and also various containment relations.



Here we have: $\mathsf{NTPP}(a, \mathsf{conv}(\mathsf{ccvia}), \mathsf{PO}(b, \mathsf{conv}(\mathsf{ccvia}))$ etc..

Convexity and Convex Hulls

A region is *convex* just in case it is equal to its own convex hull. Thus, we could define:

$$\mathsf{CONV}(x) \leftrightarrow (\mathsf{conv}(x) = x)$$

Conversely, the convex hull function can be defined by:

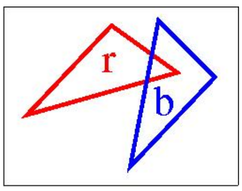
$$(y = \mathsf{conv}(x)) \leftrightarrow (\ \mathsf{CONV}(y) \wedge \mathsf{P}(x,y) \wedge \forall z[(\mathsf{CONV}(z) \wedge \mathsf{P}(x,z)) \rightarrow \mathsf{P}(y,z)]\ )$$

So, the function conv and the predicate CONV are interdefinable.

Congruence

Another very expressive spatial relation which we may wish to employ is that of congruence.

Two regions are congruent, if one can be transformed into the other by a combination of a rotation and a linear displacement and possibly also a mirror image transposition.



Exercise

Try drawing situations corresponding to the following formulae:

- $\mathsf{DC}(a,b) \wedge \mathsf{DC}(a,c) \wedge \mathsf{P}(a, \mathsf{conv}(\mathsf{sum}(b,c))$
- $\mathsf{EC}(a, \mathsf{conv}(b)) \wedge \mathsf{DC}(a,b)$
- $\mathsf{INT}(a,b,c) \wedge \mathsf{SCON}(a) \wedge \mathsf{SCON}(b) \wedge \neg\mathsf{SCON}(c)$
- $\mathsf{PO}(a,b) \wedge \mathsf{PO}(a,c) \wedge \forall x[\mathsf{C}(x,a) \rightarrow \mathsf{C}(x, \mathsf{sum}(b,c))]$
- $\exists x \exists y \exists z[\mathsf{P}(x,a) \wedge \neg\mathsf{P}(x,b) \wedge \mathsf{P}(y,a) \wedge \mathsf{P}(y,b) \wedge \mathsf{P}(z,b) \wedge \neg\mathsf{P}(z,a)]$
- $\mathsf{EC}(a,b) \wedge (\mathsf{conv}(a) = \mathsf{conv}(b))$

Knowledge Representation

Lecture KRR-11

Modes of Inference

Deduction

The form of inference we have studied so-far in this course is known as *deduction*.

An argument is deductively valid iff: given the truth of its premisses, its conclusion is necessarily true.

Induction

A mode of inference which is very common in empirical sciences is *induction*.

In this form of inference we start with a (usually large) body of specific facts (observations) and generalise from this to a universal law.

E.g. from observing many cases we might induce:

All physical bodies not subject to an external force eventually come to a state of rest.

Although supported by many facts and not contradicted by a counterexample, an inductive inference is not deductively valid.

Abduction

Abduction is the kind of reasoning where we infer from some observed fact an explanation of that fact.

Specifically, given an explanatory theory Θ and an observed fact ϕ, we may abduce α if:

$$\Theta, \alpha \models \phi$$

For abduction to be reasonable we need some way of constraining α to be a *good* explanation of ϕ.

We generally want to abduce a simple fact, not a general principle (that would be induction).

In formal logic this may be difficult; but in ordinary commonsense reasoning abduction seems to be extremely common.

Illustration of the Different Modes

1. All beans in the bag are red.
2. These beans came from the bag.
3. These beans are all red.

What are the following modes of reasoning?:

- $1,\ 2\ |\!\approx\ 3$ *deduction*
- $1,\ 3\ |\!\approx\ 2$ *abduction*
- $2,\ 3\ |\!\approx\ 1$ *induction*

Other Modes

Are there any other modes of reasoning?

Knowledge Representation

Lecture KRR-12

Multi-Valued and Fuzzy Logics

Overview

- This lecture gives a brief overview of multi-valued and fuzzy logics.
- These logics depart from classical logic in that they allow that propositions may have truth values that are intermediate between absolute truth and absolute falsity.
- We shall see that giving a semantics for multi-valued logics requires that the standard Boolean truth function interpretation of logical operators be replaced by more complex truth functions.

Classical Truth Values

In classical logic, it is assumed that every proposition is either true or false (and not both).

Thus we take as the principles of *excluded middle* and *non-contradiction* as fundamental theorems or axioms:

$$(\phi \vee \neg\phi) \qquad \neg(\phi \wedge \neg\phi)$$

We say that classical logic gives a *bi-valent* account of truth.

Degrees of Truth

One of the central ideas of multi-valued and fuzzy logics (which may be considered a type of multi-valued logic) is that certain propositions may be neither absolutely true nor absolutely false, but instead may have some intermediate truth value, which lies somewhere in between.

Such propositions typically involve vague adjectives. For instance:

- Sue is tall.
- Alfred is Bald.
- That bag is heavy.

3-Valued Logic

3-valued logic, allows each proposition to have one of three possible truth values.

As well as the usual true (**T**) and false (**F**) there is a third truth value, which I will write as: **U**.

The truth value **U** may be described as: ‘unknown’, ‘uncertain’ or ‘indeterminate’; or perhaps ‘partly true’.

3-Valued Truth Tables

Several different 3-Valued logics have been proposed, most notably those of Łukasiewicz and Kleene.

Both of these logics agree on the basic truth-tables for negation, conjunction and disjunction:

α	$\neg\alpha$
T	**F**
U	**U**
F	**T**

$(A \wedge B)$		B: **T**	**U**	**F**
A	**T**	**T**	**U**	**F**
	U	**U**	**U**	**F**
	F	**F**	**F**	**F**

$(A \vee B)$		B: **T**	**U**	**F**
A	**T**	**T**	**T**	**T**
	U	**T**	**U**	**U**
	F	**T**	**U**	**F**

3-Valued Implication Functions

Interpretation of the implication connective is more controversial.

Keene and Łukasiewicz logic give different truth tables for ‘ $\rightarrow$ ’:

Kleene

$(A \rightarrow B)$		B T	B U	B F
A	**T**	**T**	**U**	**F**
A	**U**	**T**	**U**	**U**
A	**F**	**T**	**T**	**T**

Łukasiewicz

$(A \rightarrow B)$		B T	B U	B F
A	**T**	**T**	**U**	**F**
A	**U**	**T**	**T**	**U**
A	**F**	**T**	**T**	**T**

They differ on the value of $(A \rightarrow B)$, where both A and B have the truth value **U**.

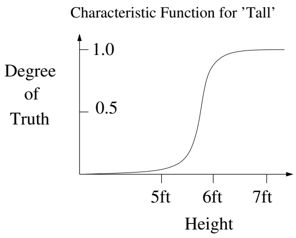
Fuzzy Logic and Fuzzy Truth Values

In the most common form of *Fuzzy Logic* the truth value of every proposition is a number in the range $[0...1]$, where:

- 1 is definitely true
- 0 is definitely false.
- 0.5 is in the middle between true and false.
- Values in $(0.5...1)$ means that the proposition is more true than false (though not completely true).
- Values in $(0...0.5)$ mean the proposition is more false than true.

Characteristic Functions for Fuzzy Sets

Fuzzy truth values are often associated with the degree of membership of an entity in a *fuzzy set*. This is often modelled by a function of some relevant measurable property.
For instance, degree of membership of a person in the set of ‘tall people’ can be modelled as a function of the height of a person:



Fuzzy Truth Functions

The truth values of propositions formed by truth functional connectives in fuzzy logic are standardly modelled by the following numerical operations:

$$\mathsf{V}(\neg A) = 1 - \mathsf{V}(A)$$

$$\mathsf{V}(A \wedge B) = Min(\mathsf{V}(A), \mathsf{V}(B))$$

$$\mathsf{V}(A \vee B) = Max(\mathsf{V}(A), \mathsf{V}(B))$$

Examples

- $\mathsf{Tall}(Alan) = 0.7$
- $\mathsf{Thin}(Alan) = 0.4$

So

$$\begin{aligned}
\neg\mathsf{Tall}(Alan) &= 1 - 0.4 &= 0.4 \\
\mathsf{Tall}(Allan) \wedge \mathsf{Thin}(Alan)) &= Min(0.7,\, 0.4) &= 0.4 \\
\neg(\mathsf{Tall}(Alan) \vee \mathsf{Thin}(Alan))) &= 1 - Max(0.7,\, 0.4) &= 0.3 \\
\mathsf{Tall}(Alan) \wedge \neg\mathsf{Thin}(Alan)) &= Min(0.7,\, (1 - 0.4)) &= 0.6
\end{aligned}$$

Very and Quite

We can also model other modifications of a proposition as fuzzy truth functions:

- $\mathsf{V}(\mathsf{Very}(\phi)) = (\mathsf{V}(\phi))^2$
- $\mathsf{V}(\mathsf{Quite}(\phi)) = (\mathsf{V}(\phi))^{1/2}$

So:

$$\begin{aligned}
\mathsf{Very}(\mathsf{Tall}(Alan)) &= 0.7^2 &= 0.49 \\
\mathsf{Quite}(\mathsf{Thin}(Alan)) &= (0.4)^{1/2} &= 0.632
\end{aligned}$$

Knowledge Representation

Lecture KRR-13

Non-Monotonic Reasoning

Monotonic *vs* Non-Monotonic Logic

Classical logic is monotonic. This means that increasing the amount of information (i.e. the number of premisses) always adds to what can be deduced. Formally we have:

$$\Gamma \vdash \phi \quad \Longrightarrow \quad \Gamma \wedge \psi \vdash \phi$$

And indeed, in the semantics for classical logics we have:

$$\Gamma \models \phi \quad \Longrightarrow \quad \Gamma \wedge \psi \models \phi$$

Conversely a proof system is non-monotonic iff:

$$\Gamma \vdash \phi \quad \not\Longrightarrow \quad \Gamma \wedge \psi \vdash \phi$$

So, adding information can make a deduction become invalid.

Motivation for Non-Monotonicity

- In commonsense reasoning we often draw conclusions that are not completely certain. We may then retract these if we get more information.
- When we communicate we tend to leave out obvious assumptions.
- In the absence of further detail we tend to associate generic descriptions with some prototype (e.g. bird $\Rightarrow$ robin).

The 'Tweety' Example

This example has been discussed endlessly in the non-monotonic reasoning literature.

Given the fact Bird(Tweety) we would (in most cases) like to infer Flies(Tweety).

We could have an axiom $\forall x[\text{Bird}(x) \rightarrow \text{Flies}(x)]$

But what if Tweety is a penguin?

We could tighten the axiom to
$\forall x[(\text{Bird}(x) \wedge \neg\text{Penguin}(x)) \rightarrow \text{Flies}(x)]$

But then if all we know is 'Bird(Tweety)' we cannot make the inference we wanted.

The Closed World Assumption

A simple form of non-monotonic reasoning is to assume that everything that is not provable is false.

So we have an additional inference rule of the form:

$$\Gamma \not\vdash \phi \quad \Longrightarrow \quad \Gamma \vdash \neg\phi$$

But this can lead to inconsistency. We generally need the restriction that ϕ must occur in Γ. But we still have problems.

Let $\Gamma = (p \vee q)$ then neither p or q follow from Γ so from the CWA we can derive $\neg p$ and $\neg q$. But the formula$(p \vee q) \wedge \neg p \wedge \neg q$ is inconsistent.

Default Logic

Proposed by Ray Reiter in 1980.

This logic is built on propositional or 1st-order logic by adding an extra inference mechanism.

Default Rules are used to specify typical (default) inferences — e.g. Birds typically fly.

We can only make inferences from default rules provided it is consistent to do so.

For example, if Tweety is a bird the by default we can conclude that he flies. If, however, we know that Tweety is a penguin (or ostrich etc.) the this inference is blocked.

Default Rules

The general form of a default rule is

$$\alpha : \beta_1, \ldots, \beta_n \ / \ \gamma$$

(The γ is often written underneath.)

This means: "*If α is true and it is consistent to believe each of the β_i (not necessarily at the same time), then one may infer (by default) γ.*

α is the prerequisite — it may sometimes be omitted.

The β_is are called 'justifications'; and γ is the conclusion.

Normal Defaults

A normal default is one where the 'justification' is the same as the conclusion:

E.g. German(x) : Drinks-Beer(x)/Drinks-Beer(x)

Thus given German(max)
we can use this default rule to infer Drinks-Beer(max)
unless we can prove ¬Drinks-Beer(max).

Normal defaults have nice computational properties.

Non-Normal Defaults

The most obvious default rules are the normal ones, where we derive a conclusion as long as that conclusion is consistent.

However, sometimes it is useful to use a justification that is different from the conclusion.

E.g. Adult(x) : (Married(x) ∧ ¬Student(x)) / Married(x)

The extra ¬Student(x) conjunct in the justification serves to block the inference when we know that x is a student.

Rules with no Prerequisite

By using rules with no prerequisite we can allow normal assumptions to be made where no information is given.

For instance if a scenario description does not mention it is raining we can assume it is not.A default Sit Calc theory might contain the rule:

: ¬Holds(Raining, s) / ¬Holds(Raining, s)

This would allow the following action precondition to be satisfied in the absence of information about rain:

$$poss(\textbf{play-football}, s) \leftarrow \neg\text{Holds}(\text{Raining}, s)$$

Default Theories

A default theory is a classical theory plus a set of default rules. Thus it can be described by pair:

$$\langle \mathcal{T}, \mathcal{D} \rangle \ ,$$

where $\mathcal{T}$ is a set of classical formulae and $\mathcal{D}$ a set of default rules.

Provability in a Default Theory

Default logics are built on top of classical logics so the notion of classical deduction can be retained in the default logic setting:

- Let $\mathcal{T} \vdash \phi$ mean that ϕ is derivable from $\mathcal{T}$ by classical (monotonic) inference.

A (naïve) non-monotonic deduction relation can be represented as follows:

Let $\mathcal{T} \ \vdash_{\mathcal{D}} \ \phi$ mean that ϕ is derivable from $\mathcal{T}$ by a combination of classical inference and the application of default rules taken from $\mathcal{D}$.

Self-Undermining Inferences

Consider the simple default theory $\langle \mathcal{T}, \mathcal{D} \rangle$, where:
$\mathcal{T} = \{R,\ P \to Q\}$
$\mathcal{D} = \{(R : \neg Q \ /\ P)\}$

Since $\neg Q$ is consistent with $\mathcal{T}$, we can apply the default rule and derive P. Then by modus ponens we immediately get Q.

But $\mathcal{T} \cup \{P\}$ now entails Q and is thus inconsistent with the justification $\neg Q$ used in the default rule.

So the applicability of the default rule is now brought into question.

Well-Founded Default Provability

To get a better-behaved entailment relation we want to block inferences that undercut the default rules used in their own derivation.

I first define a restricted entailment relation where derivations from $\mathcal{T}$ can only use defaults that are also compatible with an additional formula set $\mathcal{S}$

Let $\mathcal{T} \vdash_{(\mathcal{D}\,:\,\mathcal{S})} \phi$ mean that ϕ is derivable from $\mathcal{T}$ by a combination of classical inference and the application of default rules taken from $\mathcal{D}$ and whose justifications are consistent with $\mathcal{T} \cup \mathcal{S}$.

Extensions of Default Theories

We now characterise sets of self-consistent inferences from a default theory.

An *extension* of $\langle \mathcal{T}, \mathcal{D} \rangle$ is a set of formulae $\mathcal{E}$ such that:

1. $\mathcal{T} \subseteq \mathcal{E}$
2. If $\mathcal{E} \vdash \phi$ then $\phi \in \mathcal{E}$ (deductive closure);
3. If $\alpha \in \mathcal{E}$ and $(\alpha : \beta_1, \ldots, \beta_n \ /\ \gamma) \in \mathcal{D}$ and for $i \in \{1, \ldots, n\}$, $\neg\beta_i \notin \mathcal{E}$ then $\gamma \in \mathcal{E}$ (default closure);
4. For each $\phi \in \mathcal{E}$ we have $\mathcal{T} \vdash_{(\mathcal{D}\,:\,\mathcal{E})} \phi$ (grounded and well-founded — no undermining).

Conflicting Extensions

Suppose a default theory contains the default rules:
1) (: ¬Raining / ¬Raining)
and
2) (Wet-Washing : Raining / Raining);

and also the fact Wet-Washing.

We can apply either of the rules. But, once we have applied one, the justification of the other will be undermined.

Thus there are two distinct extensions to the theory.
One containing ¬Raining and the other containing Raining.

Is this desirable? It depends on what we want.

Validity in Default Logic

Reiter suggested that each extension can be taken as a consistent set of beliefs compatible with a default theory.

In Default Logic the notion of *valid* inference can be characterised in various different ways. The most common are the following:

Credulous entailment is defined by:

$$\langle \mathcal{T}, \mathcal{D} \rangle \models_{\mathrm{cred}} \phi$$

just in case ϕ is a member of *some* extension of $\langle \mathcal{T}, \mathcal{D} \rangle$.

Sceptical entailment is defined by:

$$\langle \mathcal{T}, \mathcal{D} \rangle \models_{\mathrm{scept}} \phi$$

just in case ϕ is a member of *every* extension of $\langle \mathcal{T}, \mathcal{D} \rangle$.

Computational Properties of Default Reasoning

Adding default reasoning to a logic can greatly increase the complexity of computing inferences.

To apply a rule $(\alpha : \beta \ /\ \gamma)$ we need to check whether β is *consistent* with the rest of the theory.

If the logic is *decidable* default reasoning will still be decidable (although usually more complex).

But if the logic is (as 1st-order logic) only *semi-decidable*, then consistency checking is *undecidable*. So default reasoning will then be fully undecidable.

Default Solution of the Frame Problem in Sit Calc

One solution of the frame problem is to assume that nothing changes unless it is forced to change by some entailment of the theory. This can be expressed in a combination of Situation Calculus and Default Logic as follows:

$$holds(\phi, s) : \ holds(\phi, result(\alpha, s)) \ / \ holds(\phi, result(\alpha, s))$$

However, we still need to ensure that the background theory we are using takes care of sementica and and causal relationships.

And, by itself, default logic does not solve the problem of *ramifications*.

Reading

To get a fuller understanding of default logic, I suggest you read the following paper:

Grigoris Antoniou (1999), *A tutorial on default logics*, ACM Computing Surveys, 31(4):337359.

DOI link: `http://doi.acm.org/10.1145/344588.344602`

You should be able to download this via the UoL library electronic resources (search for ACM Computing Surveys).

Knowledge Representation

Lecture KRR-14

Description Logic

Motivation

- AI knowledge bases often contain a large number of concept definitions, which determine the meaning of a concept in terms of other more *primitive* concepts.
- First-order logic is well suited to representing these concept definitions, but is impractical for actually computing inferences.
- We would like a representational formalism which retains enough of the expressive power of 1st-order logic to facilitate concept definitions but has better computational properties.

Relationships Between Concepts

Many types of logical reasoning depend on semantic relationships between concepts.

For instance, the necessary fact that “All dogs are mammals” could be represented in 1st-order logic as follows:

$$\forall x[\mathsf{Dog}(x) \rightarrow \mathsf{Mammal}(x)]$$

Another way of looking at the meaning of this formula is to regard it as saying that ‘Dog’ is a *subconcept* of ‘Mammal’.

This could be formalised in Description Logic as:

$$\mathsf{Dog} \sqsubseteq \mathsf{Mammal}$$

Negation, Conjunction and Disjunction

It is often useful to describe relations between concepts in terms of negation, conjunction and disjunction.

E.g. in 1st-order logic we might write:

$$\forall x[\mathsf{Bachelor}(x) \leftrightarrow (\mathsf{Male}(x) \wedge \neg\mathsf{Married}(x))]$$

In Description Logic we could simply write

$$\mathsf{Bachelor} \equiv (\mathsf{Male} \sqcap \neg\mathsf{Married})$$

Similarly we might employ a concept disjunction as follows:

$$\mathsf{Organism} \equiv (\mathsf{Plant} \sqcup \mathsf{Animal})$$

Universal and Null Concepts

For some purposes it is useful to refer to the universal concept $\top$, which is satisfied by everying, or the empty concept $\bot$, which is satisfied by nothing.

For example

$$(\mathsf{Plant} \sqcap \mathsf{Animal}) \equiv \bot$$

Or in describing a universe of physical things we might have:

$$(\mathsf{Mineral} \sqcup (\mathsf{Plant} \sqcup \mathsf{Animal})) \equiv \top$$

Instances of Concepts

In 1st-order logic we say that an individual is an instance of a concept by applying a predicate to the name of the individual. e.g. $\mathsf{Bachelor}(\mathsf{fred})$.

In description logic, concepts are just referred to by name and do not behave syntactically like predicates. Hence we introduce a special relation, which takes the place of predication. We write, e.g.:

Fred *isa* Bachelor

Rôles

We can also use relational concepts, which in DL are usually called *rôles*.

For example we can write:

Allen has-child Bob

Quantifiers

DL also allows limited form of quantification using the following (variable-free) constructs:

$$\forall r.C$$

This refers to the concept whose members are all objects, such that everything they are reletated to by r is a member of C.

e.g.

Comedian $\equiv_{def}$ (Person $\sqcap \forall$ tells-joke.Funny)

More Quantifiers

Similarly we have an existential quantifier, such that

$$\exists r.C$$

is the concept whose members are all those individuals that are related to something that is a C.

For example:

Parent $\equiv$ ($\exists$ has-child.$\top$)

Grandfather $\equiv$ Male $\sqcap$ ($\exists$ has-child.($\exists$ has-child.$\top$))

Another Example

We will define the conept of “lucky man” as a man who has a rich or beautiful wife and all his children are happy.

Another Example

Fanatics never respect each other.

Knowledge Representation

Lecture KRR-15

Propositional Resolution

Overview

The discovery of the *Resolution* inference rule was a major breakthrough in automated reasoning.

It is well-suited to computational implementation and is in most practical cases more efficient at finding proofs than previous systems.

(In terms of computational complexity theory, propositional reasoning is NP Complete, however different algorithms certainly differ in their practical efficiency.)

Propositional Resolution

Consider *modus ponens* ($\phi,\ \phi \rightarrow \psi\ \vdash \psi$) with the implication re-written as the equivalent disjunction:

$$\phi,\ \neg\phi \vee \psi\ \vdash\ \psi$$

This can be seen as a *cancellation* of ϕ with $\neg\phi$.

More generally we have the rule

$$\phi \vee \alpha,\ \neg\phi \vee \beta\ \vdash\ \alpha \vee \beta$$

This is the rule of (binary, propositional) *resolution*.

The deduced formula is called the *resolvent*.

Special Cases

As special cases of resolution — where one resolvent is not a disjunction — we have the following:

$$\phi,\ \neg\phi \vee \psi\ \vdash\ \psi$$

$$\neg\phi,\ \phi \vee \psi\ \vdash\ \psi$$

$$\neg\phi,\ \phi\ \vdash$$

In the last case an *inconsistency* has been detected.

Conjunctive Normal Form (CNF)

A *literal* is either an atomic proposition or the negation of an atomic proposition.

A *clause* is a disjunction of literals.

A CNF formula is a conjunction of clauses.

Thus a CNF formula takes the form:

$$\begin{array}{c} p_{01} \wedge\ \ldots\ \wedge p_{0m_0} \wedge \neg q_{01} \wedge\ \ldots\ \wedge \neg q_{0n_0} \wedge \\ (p_{11} \vee \ldots \vee p_{1m_1} \vee \neg q_{11} \vee \ldots \vee \neg q_{1n_1}) \wedge \\ \vdots \qquad \vdots \\ (p_{k1} \vee \ldots \vee p_{km_k} \vee \neg q_{k1} \vee \ldots \vee \neg q_{kn_k}) \end{array}$$

Set Representation of CNF

A conjunction of formulae can be represented by the set of its conjuncts.

Similarly a disjunction of literals can be represented by the set of those literals.

Thus a CNF formula can be represented as a set of sets of literals.

E.g.:

$$\{\{p\}, \{\neg q\}, \{r, s\}, \{t, \neg u, \neg v\}\}$$

represents

$$p \wedge \neg q \wedge (r \vee s) \wedge (t \vee \neg u \vee \neg v)$$

Conversion to Conjunctive Normal Form

Any propositional formula can be converted to CNF by repeatedly applying the following equivalence transforms, wherever the left hand pattern matches some sub-formula.

Rewrite $\rightarrow$:
$(\phi \rightarrow \psi) \implies (\neg\phi \vee \psi)$

Move negations inwards:

$\neg\neg\phi \implies \phi$ (Double Negation Elimination)

$\neg(\phi \vee \psi) \implies (\neg\phi \wedge \neg\psi)$ (De Morgan)

$\neg(\phi \wedge \psi) \implies (\neg\phi \vee \neg\psi)$ (De Morgan)

Distribute $\vee$ over $\wedge$:
$\phi \vee (\alpha \wedge \beta) \implies (\phi \vee \alpha) \wedge (\phi \vee \beta)$

Complete Consistency Checking for CNF

The resolution inference rule is *refutation complete* for any set of clauses.

This means that if the set is inconsistent there is a sequence of resolution inferences culminating in an inference of the form $p, \neg p \vdash$, which demonstrates this inconsistency.

If the set is consistent, repeated application of these rules to derive new clauses will eventually lead to a state where no new clauses can be derived.

Since any propositional formula can be translated into CNF, this gives a decision procedure for propositional logic. It is typically more efficient than the sequent calculus we studied earlier.

Duplicate Factoring for Clausal Formulae

If we represent a clause as disjunctive formula rather than a set of literals, there is an additional rule that must be used as well as resolution to provide a complete consistency checking procedure.

Suppose we have: $p \vee p,\ \neg p \vee \neg p$. The only resolvent of these clauses is $p \vee \neg p$. And by further resolutions we cannot derive anything but these three formulae.

The solution is to employ a factoring rule to remove duplicates:

$$\alpha \vee \phi \vee \beta \vee \phi \vee \gamma \ \vdash\ \phi \vee \alpha \vee \beta \vee \gamma$$

With the set representation, this rule is not required since by definition a *set* cannot have duplicate elements (so factoring is implicit).

Giving a Resolution Proof

In the propositional case, it is quite easy to carry out resolution proofs by hand. For example:

$$\{\{A, B, \neg C\},\ \{\neg A, D\},\ \{\neg B, E\},\ \{C, E\}\,\{\neg D, \neg A\}\,\{\neg E\}\,\}$$

Enumerate the clauses:

1. $\{A, B, \neg C\}$
2. $\{\neg A, D\}$
3. $\{\neg B, E\}$
4. $\{C, E\}$
5. $\{\neg D, \neg A\}$
6. $\{\neg E\}$

Apply resolution rules:

7. $\{\neg B\}$ from 3 & 6
8. $\{C\}$ from 4 & 6
9. $\{A, \neg C\}$ from 1 & 7
10. $\{A\}$ from 8 & 9
11. $\{\neg A\}$ from 2 & 5 (duplicate $\neg A$ deleted)
12. $\emptyset$ from 10 & 11

v

vi

Knowledge Representation

Lecture KRR-16

First-Order Resolution

1st-order Automated Reasoning

- We have seen that 1st-order reasoning is (in general) undecidable (or more precisely semi-decidable).
- Nevertheless massive effort has been spent on developing inference procedures for 1st-order logic.
- This is because 1st-order logic is a very expressive and flexible language.
- A 1st-order reasoning system that only works on simple set of formulae can sometimes be very useful.

Resolution in 1st-order Logic

Consider the following argument:

$$\text{Dog(Fido)},\ \forall x[\text{Dog}(x) \rightarrow \text{Mammal}(x)] \ \vdash\ \text{Mammal(Fido)}$$

Writing the implication as a quantified clause we have:

$$\text{Dog(Fido)},\ \forall x[\neg\text{Dog}(x) \vee \text{Mammal}(x)] \ \vdash\ \text{Mammal(Fido)}$$

If we instantiate x with Fido this is a resolution:

$$\text{Dog(Fido)},\ \neg\text{Dog(Fido)} \vee \text{Mammal(Fido)}] \ \vdash\ \text{Mammal(Fido)}$$

In 1st-order resolution we combine the instantiation and cancellation steps into a single inference rule.

Resolution without Instantiation

Resolution does not always involve instantiation. In many cases one can derive a universal consequence.

Consider the argument:

$$\forall x[\text{Dog}(x) \rightarrow \text{Mammal}(x)] \wedge \forall x[\text{Mammal}(x) \rightarrow \text{Animal}(x)] \ \vdash \ \forall x[\text{Dog}(x) \rightarrow \text{Animal}(x)]$$

Which is equivalent to :

$$\forall x[\neg\text{Dog}(x) \vee \text{Mammal}(x)] \wedge \forall x[\neg\text{Mammal}(x) \vee \text{Animal}(x)] \ \vdash \ \forall x[\neg\text{Dog}(x) \vee \text{Animal}(x)]$$

This can be derived in a single resolution step:

Mammal(x) *resolves* against $\neg$Mammal(x) for all possible values of x.

1st-order Clausal Form

To use resolution as a general 1st-order inference rule we have to convert 1st-order formulae into a clausal form similar to propositional CNF.

To do this we carry out the following sequence of transforms:

1. Eliminate $\rightarrow$ and $\leftrightarrow$ using the usual equivalences.
2. Move $\neg$ inwards using the equivalences used for CNF plus:
$$\neg\forall x[\phi] \Longrightarrow \exists x[\neg\phi]$$
$$\neg\exists x[\phi] \Longrightarrow \forall x[\neg\phi]$$
3. Rename variables so that each quantifier uses a different variable (prevents interference between quantifiers in the subsequent transforms).

4. Eliminate existential quantifiers using the *Skolemisation* transform (described later).
5. Move universal quantifiers to the left.
This is justified by the equivalences
$$\forall x[\phi] \vee \psi \ \Longrightarrow\ \forall x[\phi \vee \psi]$$
$$\forall x[\phi] \wedge \psi \ \Longrightarrow\ \forall x[\phi \wedge \psi]\ ,$$
which hold on condition that ψ does not contain the variable x.
6. Transform the *matrix* — i.e. the part of the formula following the quantifiers — into CNF using the transformations given above. (Any duplicate literals in the resulting disjunctions can be deleted.)

Skolemisation

Skolemisation is a transformation whereby existential quantifiers are replaced by constants and/or function symbols.

Skolemisation does not produce a logically equivalent formula but it does preserve consistency.

If we have a formula set $\Gamma \cup \{\exists x[\phi(x)]\}$ then this will be consistent just in case $\Gamma \cup \{\phi(\kappa)\}$ is consistent, where κ is a new arbitrary constant that does not occur in Γ or in ϕ.

Consistency is also preserved by such an instantiation in the case when $\exists x[\phi(x)]$ is embedded within arbitrary conjunctions and disjunctions (but not negations). This is because the quantifier could be moved outwards accross these connectives.

Existentials within Universals

How does Skolemisation interact with universal quantification.

Consider 1) $\forall x[\exists y[\text{Loves}(x, y)] \wedge \neg\text{Loves}(x, x)]$

How does this compare with 2) $\forall x[\text{Loves}(x, \kappa) \wedge \neg\text{Loves}(x, x)]$

From 2) we can infer $\text{Loves}(\kappa, \kappa) \wedge \neg\text{Loves}(\kappa, \kappa)]$

But this inconsistency does not follow from 1).

From 1) we can get $\exists y[\text{Loves}(\kappa, y)] \wedge \neg\text{Loves}(\kappa, \kappa)$

But then if we apply existential elimination we mush pick a *new* constant for y. So we would get, e.g.

$$\text{Loves}(\kappa, \lambda) \wedge \neg\text{Loves}(\kappa, \kappa).$$

Skolem Functions

To avoid this problem Skolem constants for existentials lying within the scope of universal quantifiers must be made to somehow vary according to possible choices for instantiations of those universals.

How can we describe something whose denotation varies depending on the value of some other variable??
By a function.

Hence Skolemisation of existentials within universals is handled by the transform:

$$\forall x_1 \ldots \forall x_n[\ldots \exists y[\phi(y)]] \implies \forall x_1 \ldots \forall x_n[\ldots \phi(f(x_1, \ldots, x_n))] \;,$$

where f is a new arbitrary function symbol.

1st-order Clausal Formulae

A 1st-order clausal formula is a disjunction of literals which may contain variables and/or Skolem constants/functions as well as ordinary constants.

All variables in a clause are universally quantified. Thus, provided we know which symbols are variables, we can omit the quantifiers. I shall use capital letters for the variables (like Prolog).

Example clauses are:

$G(a), \quad H(X, Y) \vee J(b, Y), \quad \neg P(g(X)) \vee Q(X),$

$\neg R(X, Y) \vee S(f(X, Y))$

Unification

Given two (or more) *terms* (i.e. functional expressions), *Unification* is the problem of finding a *substitution* for the variables in those terms so that the terms become identical.

A substitution my replace a variable with a constant (e.g. $X \Rightarrow c$) or functional term (e.g. $X \Rightarrow f(a)$) or with a another variable (e.g. $X \Rightarrow Y$)

A set of substitutions, θ, which unifies a set of terms is called a *unifier* for that set.

E.g. $\{(X{\Rightarrow}Z),\ (Y{\Rightarrow}Z),\ (W{\Rightarrow}g(a))\}$
is a unifier for: $\{R(X, Y, g(a)),\ R(Z, Z, W)\}$

Instances and Most General Unifiers

The result of applying a set of substitutions θ to a formula ϕ is denoted $\phi\theta$ and is called an *instance* or *instantiation* of ϕ.

If θ is a unifier for ϕ and ψ then we have $\phi\theta \equiv \psi\theta$.

There may be other unifiers θ', such that $\phi\theta' \equiv \psi\theta'$.

If for all unifiers θ' we have $\phi\theta'$ is an instance of $\phi\theta$, then $\phi\theta$ is called a *most general unifier* (or *m.g.u*) for ϕ and ψ.

An m.g.u. instantiates variables only where necessary to get a match.

If mgu$(\alpha\beta) = \theta$ but also $\alpha\theta' \equiv \beta\theta'$ then there must be some substitution θ''such that $(\alpha\theta)\theta'' \equiv \alpha\theta'$

M.g.u.s are unique *modulo* renaming variables.

An Algorithm for Computing Unifiers

There are many algorithms for computing unifiers. This is a simple re-writing algorithm.

To compute the m.g.u. of a set of expressions $\{\alpha_1, \ldots, \alpha_n\}$

Let S be the set of equations $\{\alpha_1 = \alpha_2, \ldots, \alpha_{n-1} = \alpha_n\}$

We then repeatedly apply the re-write and elimination rules given on the next slide to any suitable elements of S.

1. Identity Elimination: remove equations of the form $\alpha = \alpha$.
2. Decomposition:
   $\alpha(\beta_1, \ldots, \beta_n) = \alpha(\gamma_1, \ldots, \gamma_n) \Longrightarrow \beta_1 = \gamma_1, \ldots \beta_n = \gamma_n$.
3. Match failure: $\alpha = \beta$ or $\alpha(\ldots) = \beta(\ldots)$, where α and β are distinct constants or function symbols. There is no unifier..
4. Occurs Check failure: $X = \alpha(\ldots X \ldots)$. X cannot be equal to a term containing X. No unifier.
5. Substitution: Unless occurs check fails, replace an equation of the form $(X = \alpha)$ or $(\alpha = X)$ by $(X \Rightarrow \alpha)$ and apply the substitution $X \Rightarrow \alpha$ to all other equations in S.

After repeated application you will either reach a failure or end up with a substitution that is a unifier for all the original set of terms.

Unification Examples

Terms		Unifier
$R(X, a)$	$R(g, Y)$	$\{X \Rightarrow g,\ Y \Rightarrow a\}$
$F(X)$	$F(Y)$	$\{X \Rightarrow Y\}$ (or $\{Y \Rightarrow X\}$)
$P(X, a)$	$P(Y, f(a))$	none — $a \neq f(a)$
$T(X, f(a))$	$T(f(Z), Z)$	$\{Z \Rightarrow f(a),\ X \Rightarrow f(f(a))\}$
$T(X, a)$	$T(Z, Z)$	$\{X \Rightarrow a,\ Z \Rightarrow a\}$
$R(X, X)$	$R(a, b)$	none
$F(X)$	$F(g(a, Y))$	$X \Rightarrow g(a, Y)$.
$F(X)$	$F(g(a, X))$	none — occurs check failure.

1st-order Binary Resolution

1st-order resolution is acheived by first instantiating two clauses so that they contain complementary literals. Then an inference that is essentially the same as propositional resolution can be applied.

So, to carry out resolution on 1st-order clauses α and β we look for complementary literals $\phi \in \alpha$ and $\neg\psi \in \beta$. Such that ϕ and ψ are unifiable.

We apply the unifier to each of the clauses.

Then we can simply cancel the complementary literals and collect the remaining literals from both clauses to form the resolvent.

(We also need to avoid problems due to shared variables. See next slide.)

1st-order Binary Resolution Rule Formalised

To apply resolution to clauses α and β:

Let β' be a clause obtained by renaming variables in β so that α and β' do not share any variables.

Suppose $\alpha = \{\phi, \alpha_1, \ldots \alpha_n\}$ and $\beta' = \{\neg\psi, \beta_1, \ldots \beta_n\}$

If ϕ and ψ are unifiable a resolution can be derived.

Let θ be the m.g.u. (i.e. $\phi\theta \equiv \psi\theta$).

The resolvent of α and β is then:

$$\{\alpha_1\theta, \ldots \alpha_n\theta, \beta_1\theta, \ldots \beta_n\theta\}$$

Resolution Examples

Resolve $\{P(X),\ R(X, a)\}$ and $\{Q(Y, Z),\ \neg R(Z, Y)\}$

mgu$(R(X, a), R(Z, Y)) = \{X \Rightarrow Z,\ Y \Rightarrow a\}$

Resolvent: $\{P(Z),\ Q(a, Z)\}$

Resolve $\{A(a, X),\ H(X, Y),\ G(f(X, Y))\}$ and $\{\neg H(c, Y),\ \neg G(f(Y, g(Y)))\}$

Rename variables in 2nd clause: $\{\neg H(c, Z),\ \neg G(f(Z, g(Z)))\}$

mgu$(G(f(X, Y)),\ G(f(Z, g(Z)))) = \{X \Rightarrow Z,\ Y \Rightarrow g(Z)\}$

Resolvent: $\{A(a, Z),\ H(Z, g(Z)),\ \neg H(c, g(Z))\}$

Factoring for Refutation Compleness

Resolution by itself is *not* refutation complete.
We need to combine it with one other rule.

This is the *factoring* rule, which is the 1st-order equivalent of the deletion of identical literals in the propositional case.The rule is:
$\{\phi_1, \phi_2, \alpha_1, \ldots, \alpha_n\} \vdash \{\phi, \alpha_1\theta, \ldots, \alpha_n\theta\}$
where ϕ_1 and ϕ_2 have the same sign (both positive or both negated) and are unifiable and have θ as their m.g.u..

The combination of binary resolution and factoring inferences is *refutation complete* for clausal form 1st-order logic — i.e. from any inconsistent set of clauses these rules will eventually derive the empty clause.

v

vi

Knowledge Representation

Lecture KRR-17

Compositional Reasoning

Compositional Reasoning

Given relations $R(a, b)$ and $S(b, c)$, we may wish to know the relation between a and c.

Often this relation is constrained by the meanings of R and S.

For instance among the Allen relations we have:
During$(a, b) \wedge$ Before$(b, c) \rightarrow$ Before(a, c)

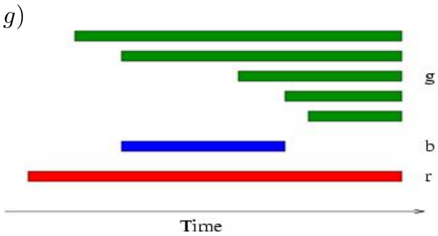
The composition of relations R and S is often written as $R; S$.

We can define: $R; S(x, y) \equiv_{def} \exists z[R(x, z) \wedge S(z, y)]$

Disjunctive Compositions

Sometimes the composition of $R(a, b)$ and $S(b, c)$ allows for a number of qualitatively different possibilities for the relation $T(a, c)$

For instance consider the case where we know During(b, r) and Ended_by(r, g)



There are 5 possible Allen relations between b and g.

Relational Partitions

In several important domains of knowledge, sets of fundamental relations $\mathcal{R} = \{R_1, \ldots, R_n\}$ have been found which are:

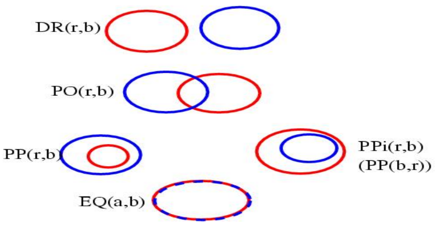
- *Mutually Exhaustive* — all pairs of objects in the domain are related by some relation in $\mathcal{R}$.
- *Pairwise Disjoint* — no two objects in the domain are related by more than one relation in $\mathcal{R}$.

Hence every pair of objects in the domain is related by exactly one relation in $\mathcal{R}$.
I shall call such a set a *Relational Partition*.

The 13 Allen relations constitute a relational partition.

The RCC-5 Relational Partition

In the domain of spatial relations there is a very general relational partition known as RCC-5 consisting of the following relations:



(This partition ignores the difference between regions touching at a boundary, which is made in RCC-8.)

Inverse and Disjunctive Relations

The inverse of a relation is defined by

$$R^{\smile}(x, y) \equiv_{def} R(y, x)$$

It will be useful to have a notation for a relation which is the disjunction of several other relations.
I shall use the notation $\{R_1, \ldots, R_n\}$, where

$$\{R_1, \ldots, R_n\}(x, y) \equiv_{def} R_1(x, y) \vee \ldots \vee R_n(x, y)$$

(For uniformity $R(x, y)$ can also be written as $\{R\}(x, y)$.)

Given a set of relations $\mathcal{R}$, the set of all disjunctive relations formed from those in $\mathcal{R}$ will be denoted by $\mathcal{R}^*$

Inverses and Disjunctive Relations in RCC-5

In RCC-5 each of the relations DR, PO, and EQ are symmetric and thus are their own inverses.

PPi is the inverse of PP and *vice versa*.

We can form arbitrary disjunctions of any of the relations. However the following disjunctions are particularly significant:

P $= \{$PP, EQ$\}$ (part)

Pi $= \{$PPi, EQ$\}$ (part inverse)

O $= \{$PO, PP, PPi, EQ$\}$ (overlap)

Relation Algebras

A *Relation Algebra* is a set of relations $\mathbf{RA}$ that is closed under: negation, disjunction, inverse and composition.

i.e. $\forall R_1, \ldots, R_n \in RA$ we have

$$\neg R_1,\ \{R_1, \ldots, R_n\},\ R_1^{\smile},\ (R_1; R_2) \in \mathbf{RA}$$

If $\mathcal{R}$ is a (finite) Relational Partition and $\mathcal{R}^*$ is closed under composition then $\mathcal{R}^*$ is a (finite) Relation Algebra; and $\mathcal{R}$ is a *basis* for that algebra.

Relation Algebras generated from a finite basis in this way have a nice computational property:

Every composition is equivalent to a disjunction of basis relations.

Composition Tables

If $\mathcal{R}^*$ is closed under composition then for every pair of relations $R_1, R_2 \in \mathcal{R}$ we can express their composition as a disjunction of relations in $\mathcal{R}$.

The compositions can then be recorded in a *Composition Table*, which allows immediate look-up of any composition.

The RCC-5 Composition Table

For RCC-5 we have the following table:

$R(a,b)$ \ $R(b,c)$	DR	PO	EQ	PP	PPi
DR	all poss	DR, PO, PP	DR	DR, PO, PP	DR
PO	DR, PO, PPi	all poss	PO	PO, PP	DR, PO, PPi
EQ	DR	PO	EQ	PP	PPi
PP	DR	DR, PO, PP	PP	PP	all poss
PPi	DR, PO, PPi	PO, PPi	PPi	O	PPi

Composing Disjunctive Relations

In general we may want to compose two disjunctive relations.

$$\{R_1, \ldots R_m\}; \{S_1, \ldots, S_n\} = \bigcup_{i=1\ldots m,\, j=1\ldots n} (R_i\ ;\ S_j)$$

Thus to compose a disjunction we:
first find the compositions of each disjunct of the first relation with each disjunct of the second relations;
then, form the disjunction of all these compositions.

Compositional (Path) Consistency

When working with a set of facts involving relations that form an RA we can use compositions as a powerful reasoning mechanism.

Wherever we have facts $R_1(a,b)$ and $R_2(b,c)$ in a logical database, we can use a composition table to look up and add some relation $R_3(a,c)$.

Where we already have information about the relation between a and c, we need to combine it with the new R_3 using the general equivalence:

$$\{\ldots, R_i, \ldots\}(x,y)\ \wedge\ \{\ldots, S_i, \ldots\}(x,y) \leftrightarrow \{\ldots, R_i, \ldots\} \cap \{\ldots, S_i, \ldots\}(x,y)$$

Compositional Completion

The rule for combining a compositional inference with existing information can be formally stated as:

$$R(x, y), S(y, z), T(x, z) \implies ((R; S) \cap T)(x, z)$$

If using this rule we find that $(R; S) \cap T) = \{\}$ we have found an *inconsistency*.

Where $T \subseteq (R; S)$, we will have $(R; S) \cap T) = T$ so the inference derives no new information.

Relational Consistency Checking Algorithm

To check consistency of a set of relational facts where the relations form an $\mathbf{RA}$, we repeatedly apply the compositional inference rule until either:

- we find an inconsistency;
- we can derive no new information from any 3 relational facts.

If we are dealing with an $\mathbf{RA}$ over a finite relational partition then this procedure must terminate.

This gives us a *decision procedure* (which runs in n^3 time, where n is the number of objects involved).

iii

iv

v

vi

Knowledge Representation

Lecture KRR-18

Uncertainty

Overview

iii

iv

v

vi

Knowledge Representation

Lecture KRR-19

Vagueness

Overview

iii

iv

v

vi

Knowledge Representation

Lecture KRR-20

Ontology and Ontologies

Overview

iii

iv

v

vi