Notes for Hong

Appendix A Extended First Order Predicate Calculus

A.1 Propositional Logic

In review:-

> Atomic propositions.
> Compound propositions, each of which has a main connective which connects its components.
>
> There are five propositional logical connectives:
>
> '~' which translates back to 'it is not the case that...'
> '&' which translates back to '... and ...'
> 'v' which translates back to '... or ...'
> '→' which translates back to 'if... then ...'

'≡' which translates back to '... if and only if ...'

A.2 Predicate Logic

In review:-

Predicate logic extends propositional logic.

'Entity-has-property' way.

The constant terms a,b,c...v are used to denote entities, the predicates A,B,C...Z are used to denote properties that these entities have, and these are put together by writing the predicate first followed by the term, for example G(b).

This technique of symbolization often may be used with English nouns, with English verbs, or with English adjectives (for example, with

'George is a runner', with
'George runs', and with
'George is running').

The propositional logical connectives can be used to extend the basic symbolization; for example, 'George runs and George is happy' might be symbolized R(g)&H(g).

There are two new logical connectives, the *Universal Quantifier* ∀x (read in English 'Whatever x you choose…') and the *Existential Quantifier* ∃x (read in English 'There is an x such that...'.). These are used for symbolizing certain English constructions. There are the variables 'w', 'x', 'y', 'z', and constants a-v. There is the *scope* of the

quantifier, and the notions of *free* and *bound* occurrences of variables.

There are polyadic predicates (relations) in addition to monadic ones. An example might be T(a,b) which might mean 'Ann is taller than Beryl', the T(?x,?y) is a binary predicate. In Description Logics, binary predicates, or binary relations, are often called **roles**.

A.3 Symbolization Templates

Using the Universal Quantifier (but not relations)

∀xT(x)

∀x~T(x)

∀x(M(x)→E(x))

∀x(T(x)∨P(x))

∀x~(T(x)&P(x))

∀x(E(x)≡W(x))

∀x(T(x)→W(x))

∀x(T(x)→~W(x))

∀x(T(x)→~U(x))

∀x(T(x)→A(x))

∀x((T(x)&O(x))→A(x))

∀x((T(x)∨D(x))→A(x))

(*note the 'or' *)

∀x(T(x)→(~R(x)→A(x)))

Universal Quantifier, with relations

$\forall xC(a,x)$

$\forall x(T(x)\&C(a,x))$

$\forall x(C(a,x)\rightarrow A(m,x))$

$\forall x(A(m,x)\rightarrow\sim C(x))$

Existential Quantifier, no relations

$\exists xO(x)$

$\exists x(B(x)\&O(x))$

$\exists x(B(x)\&\sim O(x))$

$\exists xT(x)$

$\exists x(B(x)\&(E(x)\rightarrow O(x)))$

$\exists x((T(x)\&L(x))\&\sim(R(x)\vee C(x)))$

Existential Quantifier, with relations

$\exists x\sim C(a,x)$

$\exists x\exists y(P(x)\&C(x,y))$

$\exists x(P(x)\&\exists y(C(x,y)))$

Both Quantifiers, with relations

$\forall x\exists yC(y,x)$

$\exists y\forall xC(y,x)$

$\forall y\forall xC(y,x)$

A.4 Functions, Functional Terms, and Identity

The word ‘term' in logic means 'name' and thus far we have met two kinds of terms: constants (or proper names), and variables. There also can be '*functional terms*'. For example, author(prideAndPrejudice), g(b), f(x), g(a,c), h(g(c),b) ... are all functional terms, which represent values of function applications.

Identities are true or false statements of identity between terms, so

a=b

b=7

are identities.

Functional terms often appear in identities. For example, ‘Allison’s brother’s wife’s sister is her very best friend’ might be symbolized

s(w(b(a)))=f(a)

where a=Allison, b(x)= is the brother of x, s(x)= is the sister of x, w(x)= is the wife of x, f(x)= is the very best friend of x.

Identities can be components of other more complex formulas. For example

~(b=f(a))
~∃x(b=f(x))

might symbolize ‘Bert is not Allison’s best friend’ and ‘Bert is no one’s best friend’ respectively.

A.5 Lambda Calculus

A.5.1 Plain Lambda Calculus

We need to meet lambda calculus. For lambda calculus, there are three kinds of lambda expressions:

variables,
applications, and
functions [these are sometimes called
‘abstractions’ or ‘lambda abstractions’.]

For us, a *lambda variable* is going to be exactly the same as our variables of Predicate logic ie any string of letters which starts with a lower case letter from the end of the alphabet [w,x,y,z].

An *application* consists of an opening lambda expression, a left square bracket, another lambda expression, and then a closing right square bracket. So their form is

<expression> [< expression>]

these are understood as the application of a function to an argument, so it might help to think of these as <function-expression> [<argument-expression>] For example,

x[y]

Conceptually, applications are the application of a function to an argument, just as, in trigonometry, 'cos[30]' is the application of the cosine function to the value 30, in the Lambda Calculus, 'x[y]' is the application of the function 'a' to the argument 'b'.

A function (also known as an *abstraction* or a *lambda abstraction* or an *anonymous function*) consists of the Greek letter lambda followed by a variable followed by an opening left bracket, followed by another lambda expression, and finished by a closing right bracket.. That is, they have the form

λ<variable>(<expression>)

Often it is convenient to identify or label the right hand expression, the contents of the bracket, as being the 'scope' of the abstraction. The variable itself, that goes with the lambda, is the ‘binding’ variable. Three lambda expressions which are *abstractions*, or *anonymous* functions, are

λx(y),
'λx(y[z]), and
'λw(λx(x))

The main manipulation that can be done with lambda expressions is done on those expressions which are applications, and the manipulation is that of evaluating the application. The mechanism here is very simple textual substitution. Here is an example, the application

λvariable(variable)[theArgument]

is evaluated, like any other application of an abstraction function, by taking the scope of the abstraction and substituting the argument in for all (free) occurrences of the variable in the scope. In this case the result is

theArgument

A familiar analog of lambda application evaluation is the standard Find/Replace operation of word processors. Lambda application evaluation is exactly like this. It runs through the scope of the lambda abstraction and replaces all the free occurrences of the binding variable with the argument and returns the changed contents of the scope as the result. A lambda evaluation is a textual substitution. Lambda evaluations are like macros in computer programming. Many programming systems, for example, LISP, C, and PHP have ‘macros’ (or preprocessors).

Here is a step by step evaluation (or reduction) of nested lambda applications. The example formula is

λx(λy(y)[x])[z]

We start by spotting a suitable application, with a lambda, its scope, and an argument. The outer one will do

λx (λy(y)[x]) [z]

lambda variable scope argument

Then, the argument is substituted for occurrences of the binding variable in the scope to yield

λy(y)[z]

This itself is an application and it reduces in a similar way to

z

With this example, the reduction was done on the outer reduction first. In fact, the order does not matter.

A.5.2 Convenient Generalizations Within Lambda Calculus

There are some steps that can be taken to allow Lambda Calculus to be useful for us. First, constants can be introduced as expressions. These could include the numbers 0,1,2 etc, truth values, and proper names of individuals of interest. Another expansion is to allow in already known or

predefined functions from outside. For example, the ‘successor’ function is a well-known function, it adds 1 to a natural number (thus taking 3 to 4, 117 to 118 etc.).

So, as a further example,

$\lambda x(x)$ [successor] [1]

also evaluates to 2, by a two step evaluation. It is also possible to combine lambda calculus with ordinary predicate logic. We will pass on the details here.

Appendix B Type Labels, Sorts, and Order Sorted Logic

B.1 Introduction

Separate domains.

- in geometry, there are points and lines;
- in an employment workplace there are people and there are their salaries i.e .different kinds of things.
- in database design, there regularly is analysis in terms of entitites and their relationships, and this leads to different domains; for example, a simple database for a library will have the domain of patrons (borrowers/users i.e. people) and the second domain of books.
- everywhere in computer science; for example, in any modern typed programming language there might be 'booleans', 'integers', 'strings' etc., all of these are different domains.

B.2 Type Labels I

$\forall x(M(x)\rightarrow D(x))$

$\forall x{:}m(D(x))$

The notation $\forall x{:}m$ is read 'Whatever x you choose of type m'.

So, as examples of this notation, here are some equivalences between the short form and the long form

$\forall x{:}mD(x) \equiv \forall x(M(x)\rightarrow D(x))$
$\forall x{:}aD(x) \equiv \forall x(A(x)\rightarrow D(x))$
$\forall x{:}bD(x) \equiv \forall x(B(x)\rightarrow D(x))$

This then allows us to write inferences like 'Socrates is a man, all men are wise, therefore, Socrates is wise' as follows:

$M(s), \forall x{:}mW(x) \therefore W(s)$

Similarly for the existential quantifier

$\exists x(M(x)\&D(x))$

$\exists x{:}m(D(x))$

The notation ∃x:m is read 'There is an x of type m such that ...' . =

$\exists x{:}mD(x) \equiv \exists x(M(x)\&D(x))$

$\exists x{:}aD(x) \equiv \exists x(A(x)\&D(x))$

$\exists x{:}bD(x) \equiv \exists x(B(x)\&D(x))$

So, conceptually at least, a database for a library might contain statements like

$\forall x{:}p\forall y{:}bB(x,y)$ 'Any patron is permitted to borrow any book.'

B.3 Type Labels II: Extending the notation

Say you have

$(M(x)\&D(x))\rightarrow N(x)$ (*notice the free variable x*)

as a fancier category (those things which if they are M and D are also N), then you could introduce a monadic predicate definition for this, via an equivalence e.g.

$\forall x(A(x) \equiv ((M(x)\&D(x))\rightarrow N(x)))$

and then lower case 'a' would be your type.

Types can contain relations and functions, but there needs to be only one variable with free occurrences in the type defining scope. For example, consider ‘male friends of Arthur’, this type could be represented by

(M(x)&F(x,a)) where M(x) is x is male, F(x,y) is x is a friend of y and a is a constant denoting Arthur.

B.3.1 The comprehension notation from naive set theory

There is notation similar to lambdas that would allow us to work with type terms. In naive set theory, there are many equivalences between predicate-like expressions and also many straight identity definitions between terms; what helps to bridge the two is the comprehension notation

{x: <scope>}

where the scope or body is an ordinary well formed formula, typically and normally, with free occurrences of the variable x, for example

{x: ((M(x)&D(x))→N(x))}

This notation is read 'the x such that M(x) and ...etc'. Comprehensions are intended to be terms. So formulas like identities, between comprehensions and other terms, are all well-formed formulas, e.g.

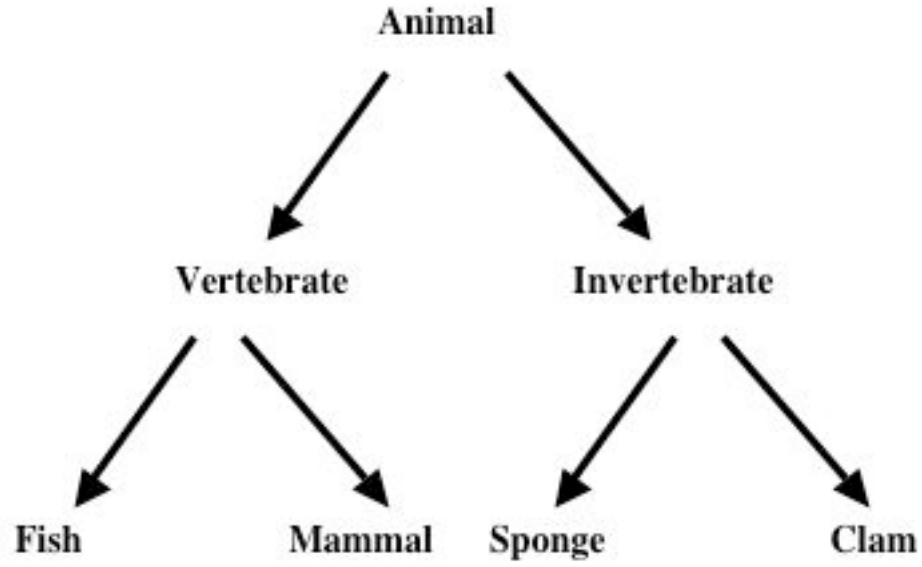
p={x: ((M(x)&D(x))→N(x))} /*this is good in naïve set theory*/

We can use comprehensions as type labels, like this, giving it a name and introduce it by a definition (using an identity not an equivalence)

$$p=\{x: ((M(x)\&D(x))\rightarrow N(x))\},$$
$$(\forall x:p)M(x)$$

B.3.2 Type labels in use

Type labels are a notational simplification. They also support shortcuts to certain inference steps. What is needed first is a syntactic addition to allow statements that relate one type to another, in particular subtypes to supertypes. A common application of relating types is with any hierarchical classification system such as genus-species in biology, or a subject classification in knowledge representation. For example,



We could certainly set about describing this by means of ordinary monadic predicates; for example,

let A(x) be the symbolization of 'x is an animal'

let V(x) be the symbolization of 'x is a vertebrate'

let I(x) be the symbolization of 'x is an Invertebrate'

then

$\forall x(V(x) \rightarrow A(x))$

$\forall x(I(x) \rightarrow A(x))$ etc.

But we could also have types (or type labels) for the predicates and write our statements

$(\forall x{:}v)A(x)$

$(\forall x{:}i)A(x)$ etc.

But then we could go one step further and allow statements between types; for example,

$\forall x(V(x) \rightarrow A(x))$

says that anything which is a vertebrate is an animal ie anything of type vertebrate is of type animal ie vertebrate is a subtype of animal; we could write this

$v<a$.

The statement 'v<a' is going to be a true or false statement in our now extended first order logic; it means

$$\forall x(V(x) \rightarrow A(x))$$

and is true or false just as that statement is true or false.

And then we can improve some of our inference steps by allowing subtypes to 'inherit' from supertypes. For examples, let us take 'St. Francis is the patron saint of all animals' as a premise

$\forall x(A(x) \rightarrow F(x))$ let F(x) mean 'St Francis is your patron saint'

We could write this

$$(\forall x{:}a)F(x)$$

and, given our classification hierarchy, we know (and could prove) that St Francis is the patron saint of Sponges etc. But we could short cut or quicken the inference steps in the proof if we allowed something like this.

$(\forall x{:}a)F(x)\ \&(s<a) \therefore (\forall x{:}s)F(x)$

that is, there is a direct inference from supertypes to subtypes; the subtypes 'inherit' from supertypes. And this is the exact logical analog of inheritance in object oriented programming.

B.3.3 Faking, or making, second order statements

First order logic has that name because the quantification is over individuals. Say we start with a predicate, M(x) meaning, perhaps 'x is a man', then if the constant 'a' signifies Arthur, the formula

M(a) says 'Arthur is a man'

M(x) is a well-formed formula, with free variable x, saying 'x is a man'

$\forall$xM(x) is a well-formed formula, saying 'Whatever x you may choose, x is a man'.

Notice with $\forall$xM(x), the quantifier, together with the x, range over all individuals. This is first order quantification, in first order logic.

There are occasions when we might want to quantify over properties. Think for one moment about identity. Say we have two items, a and b, and we worry about whether they are identical. We might reason, well, they are identical if every property that a has b has also, and every property that b has a has also i.e.

$$\forall P(P(a) \equiv P(b)) \rightarrow a=b$$

which is intended to say, 'If, whatever property P you choose, a has the property P if and only if b has the property P then a and b are identical.'

You cannot say this in first order logic because the quantification over properties is not allowed. Such a formula is allowed in second order logic. You also cannot apply one property to another in first order logic. So

M(x)

M(a)

are fine. But

M(P)

M(Q)

are not. These strings of symbols are not well-formed. Are there any occasions when we might want to apply properties to other properties? Well, think of this, say M(x) and F(x) are two properties, perhaps meaning ‘x is male’ and ‘x is female’, and T(x) and S(x) are two other properties, meaning ‘x is tall’ and ‘x is short’ and we are going to design a web form. We might say, ‘we’ll put the gender properties on the left-hand side and the height properties on the right-hand side’; and, of course, the gender properties are those of being male or of being female, and height properties are those of being tall or of being short. Were we to try to symbolize some of this, we would be looking for

Gender(Male) i.e. G(M)

G(F)

HeightProperty(Short) i.e. H(S)

S(T)

but none of this is allowed in ordinary FOL.

Let us check just what is allowed and that our analysis is correct. If **a** is Arthur then all these formulas are good in FOL

M(a) Arthur is male.

F(a) Arthur is female.

T(a) Arthur is tall.

S(a) Arthur is short.

G(a) Arthur is a gender property.*

H(a) Arthur is a height property.*

All these formulas are well-formed formulas in FOL, however the last two G(a) and H(a) do not make any sense—they are semantically unsound. The reason is the predicates or properties '…is a gender property' or '…is a height property' do not apply to individuals (like Arthur), instead they apply to other properties.

Right.

But…. with our types and type extensions we can do, or simulate, the application of one property to another. A lambda, or a set comprehension, say

{x: M(x)} and/or

λ x((M(x))

are just terms (i.e. names of individuals). So

m={x: M(x)}

is perfectly good in (extended) FOL. And if they are perfectly good, so are

G(m) and/or
G({x: M(x)})

But what are these formulas saying? Let G(?) be '? is a gender property'. then

G({x: M(x)})

says

{x: M(x)} is a gender property.

But {x: M(x)} is the concept, sort, or type 'x is a male'. So, close enough, we are applying one property to another.

We are not quantifying over properties, which is what can happen in Second Order Logic. But we are applying one property to another and effectively making a second order statement.

To review. We can introduce type labels as abbreviations for (some) quantified monadic predicates. This will make some of the formalizations

of English simpler and more natural. Then, if we allow statements relating types (ie subtype-supertype), we can make certain inferences, particularly those about classification systems, quicker and more efficient. *There is a point to note. Our logic with type labels still has just one domain or universe that the variables range over, exactly like ordinary first order logic; it is just that the type labels allow us to pick out certain elements of the universe (those satisfying the types).; many-sorted logic, to be described shortly, is different to this.*

B.4 Sorted Logic

(Many) sorted logic, with sorts or sort labels, is very similar in concept and execution to the types and type labels just discussed (in fact, many texts use the two terms interchangeably). There is, though, a conceptual difference. It is that whereas both ordinary logic and logic with type labels use one (homogenous) domain or universe, sorted logic uses a (usually stratified, heterogenous) domain which consists of two or more sub-domains. As an example, in computer science there might be a specification for a programming language that wants to use signed and unsigned integers, reals, booleans, characters, strings, etc. and wants to make such statements as 'adding a real to a real gives a real'. If such a specification is done in an unsorted logic, the monadic predicates (for 'real', 'integer', etc.) will clutter up the statements. Sorts would be much clearer here.

We will not go into this further right now.

Appendix C Description Logics

C.1 Description Logic

A Description logic is almost always (just) First Order Predicate Logic adapted to a particular purpose (see, for example, (Baader, Horrocks, & Sattler, 2007; Baader, McGuinness, & Nardi, 2003)).

Description logics typically work with sorts, types, or concepts. They often use the word 'concepts' and really that is just a synonym for types. Then they also often use either comprehension or abstraction to identify concepts. So you will see

$$\{x: ((M(x)\&D(x))\rightarrow N(x))\} \text{ or}$$
$$\lambda\, x((M(x)\&D(x))\rightarrow N(x))$$

and either of these are notations for concepts.

Just to make it easier we will use just set comprehension for concepts in this discussion, for example

$$\{x: (M(x)\&D(x))\}$$

We have already noted relations between subtypes and supertypes (or subconcepts and superconcepts). These relations are solely to do with the deductive relations between the scopes of the types. So, for example,

{x: (M(x)&D(x))} e.g. male dogs

is a subtype of

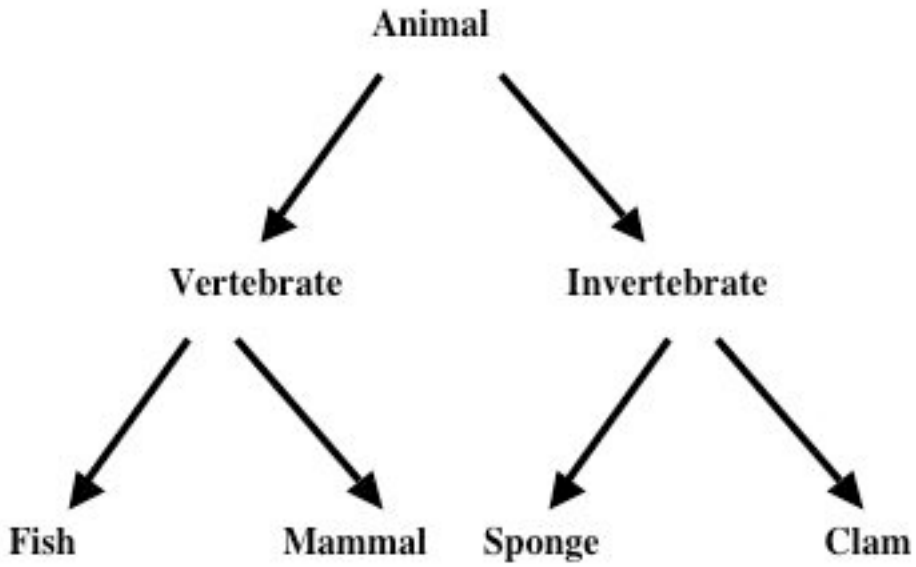
{x: (D(x))} e.g. dogs

because D(x) follows from, or is a deductive consequence of, M(x)&D(x). That the type relations are to do with deductive consequence means that it may be possible to establish the type relations, or a type hierarchy, by using a theorem prover i.e. automatically by computer.

There is a problem or consideration here. Full FOL is not decidable (that is, theoremhood cannot be established in its entirety by computer theorem proving). This means that if your types, or set constructors, are allowed to use full FOL then *you will not be able to use a theorem prover to establish the type hierarchy*. The usual solution here is to restrict the type constructors so that the result is a decidable subset of FOL and therefore theorem provers can be used.

C.2 TBox and TBox reasoning

A super simple example of a type hierarchy that you might end up with is one we have used earlier



There would be type definitions here, say

a= {x: A(x)} 'the animal type is those things with the property of being an animal'

v= {x: (A(x)&B(x))} 'the vertebrate type is those things with the property of being an animal and of having a backbone'

i= {x: (A(x)&~B(x))} 'the invertebrate type is those things with the property of being an animal and of not having a backbone'

We might also say things about the types in this type hierarchy e.g. we might say that vertebrates and invertebrates are in the second level (that

could be done with formulas like S(v) and S(i). These kinds of statements are second order statements. Some description logics call these attributes (of the types).

This type structure is often called **T-Box** and the theorem proving about it is **T-Box reasoning**.

C.3 ABox and ABox reasoning

In addition to a type structure or type hierarchy there can be, in particular areas of research, specific individual factual assertions. For example, we may determine that Ariel is a fish i.e.

F(a)

And we can put this together with our type hierarchy to reason first that Ariel is a vertebrate and then that she is an animal i.e.

V(a)
A(a)

Assertions like this take place in the **ABox** and the reasoning involved is **ABox reasoning** (which also can be done by a computer or theorem prover).

Notice that the TBox reasoning can be done in advance, pre-emptively, and at leisure, and it does not matter a lot how long it takes. In contrast, much of ABox reasoning has to be done on the spot, real time; for example, if, in the ocean you come across Ariel and note that she is a fish, then wonder whether she is in animal—answering that question by theorem proving has to be done then and there (and you won't be wanting to wait a month for the answer).

"Is a" hierarchy, inheritance of properties.

Atomic concepts (unary predicates)

Atomic roles (relations between concepts) Binary predicates (or relations)

Compound concepts.

Usually will not want to mix TBox and ABox together (not usual to allow assertions about individuals into a type hierarchy).