Implementing Semantic Theories

van Eijck, J.

DOI
10.1002/9781118882139.ch15

Publication date
2015

Document Version
Submitted manuscript

Published in
The Handbook of Contemporary Semantic Theory

Implementing Semantic Theories

Jan van Eijck[1]

Centrum Wiskunde & Informatica, Science Park 123, 1098 XG Amsterdam, The Netherlands jve@cwi.nl

ILLC, Science Park 904, 1098 XH Amsterdam, The Netherlands

A draft chapter for the Wiley-Blackwell *Handbook of Contemporary Semantics — second edition*, edited by Shalom Lappin and Chris Fox. This draft formatted on 8th April 2014.

1 Introduction

What is a semantic theory, and why is it useful to implement semantic theories?

In this chapter, a semantic theory is taken to be a collection of rules for specifying the interpretation of a class of natural language expressions. An example would be a theory of how to handle quantification, expressed as a set of rules for how to interpret determiner expressions like *all*, *all except one*, *at least three but no more than ten*.

It will be demonstrated that implementing such a theory as a program that can be executed on a computer involves much less effort than is commonly thought, and has greater benefits than most linguists assume. Ideally, this Handbook should have example implementations in all chapters, to illustrate how the theories work, and to demonstrate that the accounts are fully explicit.

What makes a semantic theory easy or hard to implement?

What makes a semantic theory easy to implement is formal explicitness of the framework in which it is stated. Hard to implement are theories stated in vague frameworks, or stated in frameworks that elude explicit formulation because they change too often or too quickly. It helps if the semantic theory itself is stated in more or less formal terms.

Choosing an implementation language: imperative versus declarative

Well-designed implementation languages are a key to good software design, but while many well designed languages are available, not all kinds of language are equally suited for implementing semantic theories.

Programming languages can be divided very roughly into imperative and declarative. Imperative programming consists in specifying a sequence of assignment actions, and reading off computation results from registers. Declarative programming consists in defining functions or predicates and executing these definitions to obtain a result.

Recall the old joke of the computer programmer who died in the shower? He was just following the instructions on the shampoo bottle: “Lather, rinse, repeat.” Following a sequence of instructions to the letter is the essence of imperative programming. The joke also has a version for functional programmers. The definition on the shampoo bottle of the functional programmer runs:

```
wash = lather : rinse : wash
```

This is effectively a definition by co-recursion (like definition by recursion, but without a base case) of an infinite stream of lathering followed by rinsing followed by lathering followed by . . . .

To be suitable for the representation of semantic theories, an implementation language has to have good facilities for specifying *abstract data types*. The key feature in specifying abstract data types is to present a precise description of that data type without referring to any concrete representation of the objects of that datatype and to specify operations on the data type without referring to any implementation details.

This abstract point of view is provided by many-sorted algebras. Many sorted algebras are specifications of abstract datatypes. Most state-of-the art functional programming languages excel here. See below. An example of an abstract data type would be the specification of a grammar as a list of context free rewrite rules, say in Backus Naur form (BNF).

Logic programming or functional programming: trade-offs

First order predicate logic can be turned into a computation engine by adding SLD resolution, unification and fixpoint computation. The result is called *datalog*. SLD resolution is *L*inear resolution with a *S*election function for *D*efinite sentences. Definite sentences, also called Horn clauses, are clauses with exactly one positive literal. An example:

$$\text{father}(x) \vee \neg\text{parent}(x) \vee \neg\text{male}(x).$$

This can be viewed as a definition of the predicate *father* in terms of the predicates *parent* and *male*, and it is usually written as a reverse implication, and using a comma:

$$\text{father}(x) \leftarrow \text{parent}(x), \text{male}(x).$$

To extend this into a full fledged programming paradigm, backtracking and cut (an operator for pruning search trees) were added (by Alain Colmerauer and Robert Kowalski, around 1972). The result is *Prolog*, short for *programmation logique*. Excellent sources of information on Prolog can be found at `http://www.learnprolognow.org/` and `http://www.swi-prolog.org/`.

Pure lambda calculus was developed in the 1930s and 40s by the logician Alonzo Church, as a foundational project intended to put mathematics on a firm basis of 'effective procedures'. In the system of pure lambda calculus, *everything* is a function. Functions can be applied to other functions to obtain values by a process of application, and new functions can be constructed from existing functions by a process of lambda abstraction.

Unfortunately, the system of pure lambda calculus admits the formulation of Russell's paradox. Representing sets by their characteristic functions (essentially procedures for separating the members of a set from the non-members), we can define

$$r = \lambda x \cdot \neg(x\ x).$$

Now apply r to itself:

$$
\begin{aligned}
r\ r &= (\lambda x \cdot \neg(x\ x))(\lambda x \cdot \neg(x\ x)) \\
&= \neg((\lambda x \cdot \neg(x\ x))(\lambda x \cdot \neg(x\ x))) \\
&= \neg(r\ r).
\end{aligned}
$$

So if $(r\ r)$ is true then it is false and vice versa. This means that pure lambda calculus is not a suitable foundation for mathematics. However, as Church and Turing realized, it is a suitable foundation for computation. Elements of lambda calculus have found their way into a number of programming languages such as Lisp, Scheme, ML, Caml, Ocaml, and Haskell.

In the mid-1980s, there was no "standard" non-strict, purely-functional programming language. A language-design committee was set up in 1987, and the Haskell language is the result. Haskell is named after Haskell B. Curry, a logician who has the distinction of having *two* programming languages named after him, *Haskell* and *Curry*. For a famous defense of functional programming the reader is referred to Hughes (1989). A functional language has *non-strict evaluation* or *lazy evaluation* if evaluation of expressions stops 'as soon as possible'. In particular, only arguments that are necessary for the outcome are computed, and only as far as necessary. This makes it possible to handle infinite data structures such as infinite lists. We will use this below to represent the infinite domain of natural numbers.

A declarative programming language is better than an imperative programming language for implementing a description of a set of semantic rules. The two main declarative programming styles that are considered suitable for implementating computational semantics are logic programming and functional programming. Indeed, computational paradigms that emerged in computer science, such as unification and proof search, found their way into semantic theory, as basic feature value computation mechanisms and as resolution algorithms for pronoun reference resolution.

If unification and first order inference play an important role in a semantic theory, then a logic programming language like Prolog may seem a natural choice as an implementation language. However, while unification and proof search for definite clauses constitute the core of logic programming (there is hardly more to Prolog than these two ingredients), functional programming encompasses the whole world of abstract datatype definition and polymorphic typing. As we will demonstrate below, the key ingredients of logic programming are easily expressed in Haskell, while Prolog is not very suitable for expressing data abstraction. Therefore, in this chapter we will use Haskell rather than Prolog as our implementation language. For a textbook on computational semantics that uses Prolog, we refer to Blackburn & Bos (2005). A recent computational semantics textbook that uses Haskell is Eijck & Unger (2010).

Modern functional programming languages such as Haskell are in fact implementations of typed lambda calculus with a flexible type system. Such languages have polymorphic types, which means that functions and operations can apply generically to data. E.g., the operation that joins two lists has as its only requirement that the lists are of the same type a — where a can be the type of integers, the type of characters, the type of lists of characters, or any other type — and it yields a result that is again a list of type a.

This chapter will demonstrate, among other things, that implementing a Montague style fragment in a functional programming language with flexible types is a breeze: Montague's underlying representation language is typed lambda calculus, be it without type flexibility, so Montague's specifications of natural language fragments in PTQ Montague (1973) and UG Montague (1974b) are in fact already specifications of functional programs. Well, almost.

Unification versus function composition in logical form construction

If your toolkit has just a hammer in it, then everything looks like a nail. If your implementation language has built-in unification, it is tempting to use unification for the composition of expressions that represent meaning. The Core Language Engine Alshawi (1992); Alshawi & Eijck (1989) uses unification to construct logical forms.

For instance, instead of combining noun phrase interpretations with verb phrase interpretations by means of functional composition, in a Prolog implementation a verb phrase interpretation typically has a Prolog variable `X` occupying a `subjVal` slot, and the noun phrase interpretation typically unifies with the `X`. But this approach will not work if the verb phrase contains more than one occurrence of `X`. Take the translation of *No one was allowed to pack and leave*. This does not mean the same as *No one was allowed to pack and no one was allowed to leave*. But the confusion of the two is hard to avoid under a feature unification approach.

Theoretically, function abstraction and application in a universe of higher order types are a much more natural choice for logical form construction. Using an implementation language that is based on type theory and function abstraction makes it particularly easy to implement the elements of semantic processing of natural language, as we will demonstrate below.

Literate Programming

This Chapter is written in so-called literate programming style. Literate programming, as advocated by Donald Knuth in Knuth (1992), is a way of writing computer programs where the first and foremost aim of the presentation of a program is to make it easily accessible to humans. Program and documentation are in a single file. In fact, the program source text is extracted from the LaTeX source text of the chapter. Pieces of program source text are displayed as in the following Haskell module declaration for this Chapter:

```
module IST where

import Data.List
import Data.Char
import System.IO
```

This declares a module called *IST*, for "Implementing a Semantic Theory", and imports the Haskell library with list processing routines called *Data.List*, the library with character processing functions *Data.Char*, and the input-output routines library *System.IO*.

We will explain most programming constructs that we use, while avoiding a full blown tutorial. For tutorials and further background on programming in Haskell we refer the reader to `www.haskell.org`, and to the textbook Eijck & Unger (2010).

You are strongly encouraged to install the Haskell Platform on your computer, download the software that goes with this chapter from internet address `https://github.com/janvaneijck/ist`, and try out the code for yourself. The advantage of developing fragments with the help of a computer is that interacting with the code gives us feedback on the clarity and quality of our formal notions.

The role of models in computational semantics

If one looks at computational semantics as an enterprise of constructing logical forms for natural language sentences to express their meanings, then this may seem a rather trivial exercise, or as Stephen Pulman once phrased it, an "exercise in typesetting". "John loves Mary" gets translated into $L(j,m)$, and so what? The point is that $L(j,m)$ is a predication that can be checked for truth in an appropriate formal model. Such acts of model checking are what computational semantics is all about. If one implements computational semantics, one implements appropriate models for semantic interpretation as well, plus the procedures for model checking that make the computational engine tick. We will illustrate this with the examples in this Chapter.

2 Direct Interpretation or Logical Form?

In Montague style semantics, there are two flavours: use of a logical form language, as in PTQ Montague (1973) and UG Montague (1974b), and direct semantic interpretation, as in EAAFL Montague (1974a).

To illustrate the distinction, consider the following BNF grammar for generalized quantifiers:

$$\text{Det} ::= \text{Every} \mid \text{All} \mid \text{Some} \mid \text{No} \mid \text{Most}.$$

The data type definition in the implementation follows this to the letter:

```
data Det = Every | All | Some | No | Most
  deriving Show
```

Let D be some finite domain. Then the interpretation of a determiner on this domain can be viewed as a function of type $\mathcal{P}D \to \mathcal{P}D \to \{0,1\}$. In Montague style, elements of D have type e and the type of truth values is denoted t, so this becomes: $(e \to t) \to (e \to t) \to t$. Given two subsets p, q of D, the determiner relation does or does not hold for these subsets. E.g., the quantifier relation All holds between two sets p and q iff $p \subseteq q$. Similarly the quantifier relation Most holds between two finite sets p and q iff $p \cap q$ has more elements than $p - q$. Let's implement this.

Direct interpretation

A direct interpretation instruction for "All" for a domain of integers (so now the role of e is played by `Int`) is given by:

```
intDET :: [Int] -> Det
                -> (Int -> Bool) -> (Int -> Bool) -> Bool
intDET domain All =  \ p q ->
  filter (\x -> p x && not (q x)) domain == []
```

Here, `[]` is the empty list. The type specification says that *intDET* is a function that takes a list of integers, next a determiner *Det*, next an integer property, next another integer property, and yields a boolean (*True* or *False*). The function definition for *All* says that *All* is interpreted as the relation between properties p and q on a *domain* that evaluates to *True* iff the set of objects in the domain that satisfy p but not q is empty.

Let's play with this. In Haskell the property of being greater than some number n is expressed as `(> n)`. A list of integers can specified as `[n..m]`. So here goes:

```
*IST> intDET [1..100] All (> 2) (> 3)
```

```
False
*IST> intDET [1..100] All (> 3) (> 2)
True
```

All numbers in the range 1..100 that are greater that 2 are also greater than 3 evaluates to *False*, all numbers s in the range 1..100 that are greater that 3 are also greater than 2 evaluates to *True*. We can also evaluate on infinite domains. In Haskell, if n is an integer, then `[n..]` gives the infinite list of integer numbers starting with n, in increasing order. This gives:

```
IST> intDET [1..] All (> 2) (> 3)
False
*IST> intDET [1..] All (> 3) (> 2)
...
```

The second call does not terminate, for the model checking procedure is dumb: it does not 'know' that the domain is enumerated in increasing order. By the way, you *are* trying out these example calls for yourself, aren't you?

A direct interpretation instruction for "Most" is given by:

```
intDET domain Most = \ p q ->
 let
   xs = filter (\x -> p x && not (q x)) domain
   ys = filter (\x -> p x && q x) domain
 in length ys > length xs
```

This says that *Most* is interpreted as the relation between properties p and q that evaluates to *True* iff the set of objects in the domain that satisfy both p and q is larger than the set of objects in the domain that satisfy p but not q. Note that this implementation will only work for finite domains.

Translation into logical form

To contrast this with translation into logical form, we define a datatype for formulas with generalized quantifiers.

Building blocks that we need for that are *names* and *identifiers* (type `Id`), which are pairs consisting of a name (a string of characters) and an integer index.

```
type Name = String
data Id  = Id Name Int deriving (Eq,Ord)
```

What this says is that we will use *Name* is a synonym for *String*, and that an object of type *Id* will consist of the identifier *Id* followed by a *Name* followed by an *Int*. In Haskell, *Int* is the type for fixed-length integers. Here are some examples of identifiers:

```
ix = Id "x" 0
iy = Id "y" 0
iz = Id "z" 0
```

From now on we can use *ix* for `Id "x" 0`, and so on. Next, we define terms. Terms are either variables or functions with names and term arguments. First in BNF notation:

$$t ::= v_i \mid f_i(t, \ldots, t).$$

The indices on variables v_i and function symbols f_i can be viewed as names. Here is the corresponding data type:

```
data Term = Var Id | Struct Name [Term] deriving (Eq,Ord)
```

Some examples of variable terms:

```
x    = Var ix
y    = Var iy
z    = Var iz
```

An example of a constant term (a function without arguments):

```
zero :: Term
zero  = Struct "zero" []
```

Some examples of function symbols:

```
s     = Struct "s"
t     = Struct "t"
u     = Struct "u"
```

Function symbols can be combined with constants to define so-called *ground terms* (terms without occurrences of variables). In the following, we use $s[\,]$ for the successor function.

```
one   = s[zero]
two   = s[one]
three = s[two]
four  = s[three]
five  = s[four]
```

The function *isVar* checks whether a term is a variable; it uses the type *Bool* for Boolean (true or false). The type specification `Term -> Bool` says that *isVar* is a classifier of terms. It classifies the the terms that start with `Var` as variables, and all other terms as non-variables.

```
isVar :: Term -> Bool
isVar (Var _) = True
isVar _       = False
```

The function *isGround* checks whether a term is a ground term (a term without occurrences of variables); it uses the Haskell primitives *and* and *map*, which you should look up in a Haskell tutorial if you are not familiar with them.

```
isGround :: Term -> Bool
isGround (Var _) = False
isGround (Struct _ ts) = and (map isGround ts)
```

This gives (you should check this for yourself):

```
*IST> isGround zero
True
*IST> isGround five
True
*IST> isGround (s[x])
False
```

The functions *varsInTerm* and *varsInTerms* give the variables that occur in a term or a term list. Variable lists should not contain duplicates; the function *nub* cleans up the variable lists. If you are not familiar with *nub*, *concat* and function composition by means of $\cdot$, you should look up these functions in a Haskell tutorial.

```
varsInTerm :: Term -> [Id]
varsInTerm (Var i)       = [i]
varsInTerm (Struct _ ts) = varsInTerms ts

varsInTerms :: [Term] -> [Id]
varsInTerms = nub . concat . map varsInTerm
```

We are now ready to define formulas from atoms that contain lists of terms. First in BNF:

$$\phi ::= A(t, \ldots, t) \mid t = t \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid Q_v\phi\phi.$$

Here $A(t, \ldots, t)$ is an atom with a list of term arguments. In the implementation, the data-type for formulas can look like this:

```
data Formula = Atom Name [Term]
             | Eq Term Term
             | Not Formula
             | Cnj [Formula]
             | Dsj [Formula]
             | Q Det Id Formula Formula
             deriving Show
```

Equality statements `Eq Term Term` express identities $t_1 = t_2$. The `Formula` data type defines conjunction and disjunction as lists, with the intended meaning that `Cnj fs` is true iff all formulas in `fs` are true, and that `Dsj fs` is true iff at least one formula in `fs` is true. This will be taken care of by the truth definition below.

Before we can use the data type of formulas, we have to address a syntactic issue. The determiner expression is translated into a logical form construction recipe, and this recipe has to make sure that variables bound by a newly introduced generalized quantifier are bound properly. The definition of the `fresh` function that takes care of this can be found in the appendix. It is used in the translation into logical form for the quantifiers:

```
lfDET :: Det ->
         (Term -> Formula) -> (Term -> Formula) -> Formula
lfDET All p q  = Q All i (p (Var i)) (q (Var i)) where
   i = Id "x" (fresh [p zero, q zero])
lfDET Most p q = Q Most i (p (Var i)) (q (Var i)) where
   i = Id "x" (fresh [p zero, q zero])
lfDET Some p q = Q Some i (p (Var i)) (q (Var i)) where
   i = Id "x" (fresh [p zero, q zero])
lfDET No p q = Q No i (p (Var i)) (q (Var i)) where
   i = Id "x" (fresh [p zero, q zero])
```

Note that the use of a fresh index is essential. If an index `i` is not fresh, this means that it is used by a quantifier somewhere inside `p` or `q`, which gives a risk that if these expressions of type `Term -> Formula` are applied to `Var i`, occurrences of this variable may get bound by the wrong quantifier expression.

Of course, the task of providing formulas of the form *All* $v\ \phi_1\phi_2$ or the form *Most* $v\ \phi_1\phi_2$ with the correct interpretation is now shifted to the truth definition for the logical form language. We will turn to this in the next Section.

3 Model Checking Logical Forms

The example formula language from Section 2 is first order logic with equality and the generalized quantifier *Most*. This is a genuine extension of first order logic with equality, for it is proved in Barwise & Cooper (1981) that *Most* is not expressible in first order logic.

Once we have a logical form language like this, we can dispense with extending this to a higher order typed version, and instead use the implementation language to construct the higher order types.

Think of it like this. For any type a, the implementation language gives us properties (expressions of type $a \to \text{Bool}$), relations (expressions of type $a \to a \to \text{Bool}$), higher order relations (expressions of type $(a \to \text{Bool}) \to (a \to \text{Bool}) \to \text{Bool}$), and so on. Now replace the type of Booleans with that of logical forms or formulas (call it F), and the type a with that of terms (call it T). Then the type $T \to F$ expresses an LF property, the type $T \to T \to F$ an LF relation, the type $(T \to F) \to (T \to F) \to F$ a higher order relation, suitable for translating generalized quantifiers, and so on.

For example, the LF translation of the generalized quantifier *Most* in Section 2, produces an expression of type $(T \to F) \to (T \to F) \to F$.

Tarski's famous truth definition for first order logic (Tarski, 1956) has as key ingredients variable assignments, interpretations for predicate symbols, and interpretations for function symbols, and proceeds by recursion on the structure of formulas.

A domain of discourse D together with an interpretation function I that interprets predicate symbols as properties or relations on D, and function symbols as functions on D, is called a *first order model*.

In our implementation, we have to distinguish between the interpretation for the predicate letters and that for the function symbols, for they have different types:

```
type Interp a  = Name -> [a] -> Bool
type FInterp a = Name -> [a] -> a
```

These are polymorphic declarations: the type `a` can be anything. Suppose our domain of entities consists of integers. Let us say we want to interpret on the domain of the natural numbers. Then the domain of discourse is infinite. Since our implementation language has non-strict evaluation, we can handle infinite lists. The domain of discourse is given by:

```
naturals :: [Integer]
naturals = [0..]
```

The type `Integer` is for integers of arbitrary size. Other domain definitions are also possible. Here is an example of a finite number domain, using the fixed size data type `Int`:

```
numbers :: [Int]
numbers = [minBound..maxBound]
```

Let V be the set of variables of the language. A function $g \colon V \to D$ is called a *variable assignment* or *valuation*.

Before we can turn to evaluation of formulas, we have to construct valuation functions of type `Term -> a`, given appropriate interpretations for function symbols, and given an assignment to the variables that occur in terms.

A variable assignment, in the implementation, is a function of type `Id -> a`, where `a` is the type of the domain of interpretation. The term lookup function takes a function symbol interpretatiomn (type `FInterp a`) and variable assigment (type `Id -> a`) as inputs, and constructs a term assignment (type `Term -> a`), as follows.

```
tVal :: FInterp a -> (Id -> a) -> Term -> a
tVal fint g (Var v)         = g v
tVal fint g (Struct str ts) =
           fint str (map (tVal fint g) ts)
```

tVal computes a value (an entity in the domain of discourse) for any term, on the basis of an interpretation for the function symbols and an assigment of entities to the variables. Understanding how this works is one of the keys to understanding the truth definition for first order predicate logic, as it is explained in textbooks of logic. Here is that explanation once more:

- If the term is a variable, *tVal* borrows its value from the assignment g for variables.
- If the term is a function symbol followed by a list of terms, then *tVal* is applied recursively to the term list, which gives a list of entities, and next the interpretation for the function symbol is used to map this list to an entity.

Example use: `fint1` gives an interpretation to the function symbol `s` while `(\ _ -> 0)` is the anonymous function that maps any variable to 0. The result of applying this to the term *five* (see the definition above) gives the expected value:

```
*IST> tVal fint1 (\ _ -> 0) five
5
```

The truth definition of Tarski assumes a relation interpretation, a function interpretation and a variable assigment, and defines truth for logical form expression by recursion on the structure of the expression.

Given a structure with interpretation function $M=(D,I)$, we can define a valuation for the predicate logical formulas, provided we know how to deal with the values of individual variables.

Let g be a variable assignment or valuation. We use $g[v := d]$ for the valuation that is like g except for the fact that v gets value d (where g might have assigned a different value). For example, let $D = \{1, 2, 3\}$ be the domain of discourse, and let $V = \{v_1, v_2, v_3\}$. Let g be given by $g(v_1) = 1, g(v_2) = 2, g(v_3) = 3$. Then $g[v_1 := 2]$ is the valuation that is like g except for the fact that v_1 gets the value 2, i.e. the valuation that assigns 2 to v_1, 2 to v_2, and 3 to v_3.

Here is the implementation of $g[v := d]$:

```
change :: (Id -> a) -> Id -> a -> Id -> a
change g v d = \ x -> if x == v then d else g x
```

Let $M = (D, I)$ be a model for language L, i.e., D is the domain of discourse, I is an interpretation function for predicate letters and function symbols. Let g be a variable assignment for L in M. Let F be a formula of our logical form language.

Now we are ready to define the notion $M \models_g F$, for F is true in M under assignment g, or: g satisfies F in model M. We assume P is a one-place predicate letter, R is a two-place predicate letter, S is a three-place predicate letter. Also, we use $[\![t]\!]^I_g$ as the term interpretation of t under I and g. With this notation, Tarski's truth definition can be stated as follows:

$$
\begin{array}{lll}
M \models_g Pt & \text{iff} & [\![t]\!]^I_g \in I(P) \\
M \models_g R(t_1, t_2) & \text{iff} & ([\![t_1]\!]^I_g, [\![t_2]\!]^I_g) \in I(R) \\
M \models_g S(t_1, t_2, t_3) & \text{iff} & ([\![t_1]\!]^I_g, [\![t_2]\!]^I_g, [\![t_3]\!]^I_g) \in I(S) \\
M \models_g (t_1 = t_2) & \text{iff} & [\![t_1]\!]^I_g = [\![t_2]\!]^I_g \\
M \models_g \neg F & \text{iff} & \text{it is not the case that } M \models_g F. \\
M \models_g (F_1 \wedge F_2) & \text{iff} & M \models_g F_1 \text{ and } M \models_g F_2 \\
M \models_g (F_1 \vee F_2) & \text{iff} & M \models_g F_1 \text{ or } M \models_g F_2 \\
M \models_g QvF_1F_2 & \text{iff} & \{d \mid M \models_{g[v:=d]} F_1\} \text{ and } \{d \mid M \models_{g[v:=d]} F_2\} \\
 & & \text{are in the relation specified by } Q
\end{array}
$$

What we have presented just now is a recursive definition of truth for our logical form language. The 'relation specified by Q' in the last clause refers to the generalized quantifier interpretations for *all*, *some*, *no* and *most*. Here is an implementation of quantifiers are relations:

```
qRel :: Eq a => Det -> [a] -> [a] -> Bool
qRel All xs ys = all (\x -> elem x ys) xs
qRel Some xs ys = any (\x -> elem x ys) xs
qRel No xs ys = not (qRel Some xs ys)
qRel Most xs ys =
  length (intersect xs ys) > length (xs \\ ys)
```

If we evaluate closed formulas — formulas without free variables — the assignment g is irrelevant, in the sense that any g gives the same result. So for closed formulas F we can simply define $M \models F$ as: $M \models_g F$ for some variable assignment g. But note that the variable assignment is still crucial for the truth definition, for the property of being closed is not inherited by the components of a closed formula.

Let us look at how to implement an evaluation function. It takes as its first argument a domain, as its second argument a predicate interpretation function, as its third argument a function interpretation function, as its fourth argument a variable assignment, as its fifth argument a formula, and it yields a truth value. It is defined by recursion on the structure of the formula. The type of the evaluation function `eval` reflects the above assumptions.

```
eval :: Eq a   =>
    [a]        ->
    Interp a   ->
    FInterp a  ->
    (Id -> a)  ->
    Formula    -> Bool
```

The evaluation function is defined for all types `a` that belong to the class `Eq`. The assumption that the type `a` of the domain of evaluation is in `Eq` is needed in the evaluation clause for equalities. The evaluation function takes a universe (represented as a list, `[a]`) as its first argument, an interpretation function for relation symbols (`Interp a`) as its second argument, an interpretation function for function symbols as its third argument, a variable assignment (`Id -> a`) as its fourth argument, and a formula as its fifth argument. The definition is by structural recursion on the formula:

```
eval domain i fint = eval' where
  eval' g (Atom str ts)   = i str (map (tVal fint g) ts)
  eval' g (Eq   t1 t2)    = tVal fint g t1 == tVal fint g t2
  eval' g (Not  f)        = not (eval' g f)
  eval' g (Cnj fs)        = and (map (eval' g) fs)
  eval' g (Dsj fs)        = or  (map (eval' g) fs)
  eval' g (Q det v f1 f2) = let
     restr = [ d | d <- domain, eval' (change g v d) f1 ]
     body  = [ d | d <- domain, eval' (change g v d) f2 ]
   in qRel det restr body
```

This evaluation function can be used to check the truth of formulas in appropriate domains. The domain does not have to be finite. Suppose we want to check the truth of "There are even natural numbers". Here is the formula:

```
form0 = Q Some ix (Atom "Number" [x]) (Atom "Even" [x])
```

We need an interpretation for the predicates "Number" and "Even". We also throw in an interpretation for "Less than":

```
int0 :: Interp Integer
int0 "Number" = \[x] -> True
int0 "Even"   = \[x] -> even x
int0 "Less_than" = \[x,y] -> x < y
```

Note that relates language (strings like "Number", "Even") to predicates on a model (implemented as Haskell functions). So the function `int0` is part of the bridge between language and the world (or: between language and the model under consideration).

For this example, we don't need to interpret function symbols, so any function interpretation will do. But for other examples we want to give names to certain numbers, using the constants "zero", "s", "plus", "times". Here is a suitable term interpretation function for that:

```
fint0 :: FInterp Integer
fint0 "zero"  []    = 0
fint0 "s"     [i]   = succ i
fint0 "plus"  [i,j] = i + j
fint0 "times" [i,j] = i * j
```

Again we see a distinction between syntax (expressions like "plus" and "times") and semantics (Haskell operations like + and *).

```
*IST> eval naturals int0 fint0 (\ _ -> 0) form0
True
```

This example uses a variable assigment `\ _ -> 0` that maps any variable to 0.

Now suppose we want to evaluate the following formula:

```
form1 = Q All ix (Atom "Number" [x])
          (Q Some iy (Atom "Number" [y])
                     (Atom "Less_than" [x,y]))
```

This says that for every number there is a larger number, which as we all know is true on the natural numbers. But this fact cannot be established by model checking. The following computation does not halt:

```
*IST> eval naturals int0 fint0 (\ _ -> 0) form1
...
```

This illustrates that model checking on the natural numbers is undecidable. Still, many useful facts can be checked, and new relations can be defined in terms of a few primitive ones.

Suppose we want to define the relation "divides". A natural number x divides a natural number y if there is a number z with the property that $x * z = y$. This is easily defined, as follows:

```
divides :: Term -> Term -> Formula
divides m n = Q Some iz (Atom "Number" [z])
              (Eq n (Struct "times" [m,z]))
```

This gives:

```
*IST> eval naturals int0 fint0 (\ _ -> 0) (divides two four)
True
```

The process of defining truth for expressions of natural language is similar to that of evaluating formulas in mathematical models. The differences are that the models may have more internal structure than mathematical domains, and that substantial vocabularies need to be interpreted.

Interpretation of Natural Language Fragments

Where in mathematics it is enough to specify the meanings of 'less than', 'plus' and 'times', and next define notions like 'even', 'odd', 'divides', 'prime', 'composite', in terms of these primitives, in natural language understanding there is no such privileged core lexicon. This means we need interpretations for all non-logical items in the lexicon of a fragment.

To give an example, assume that the domain of discourse is a finite set of entities. Let the following data type be given.

```
data Entity = A | B | C | D | E | F | G
            | H | I | J | K | L | M
     deriving (Eq,Show,Bounded,Enum)
```

Now we can define entities as follows:

```
entities :: [Entity]
entities =  [minBound..maxBound]
```

Now, proper names will simply be interpreted as entities.

```
alice, bob, carol :: Entity
alice      = A
bob        = B
carol      = C
```

Common nouns such as *girl* and *boy* as well as intransitive verbs like *laugh* and *weep* are interpreted as properties of entities. Transitive verbs like *love* and *hate* are interpreted as relations between entities.

Let's define a type for predications:

```
type Pred a = [a] -> Bool
```

Some example properties:

```
girl, boy :: Pred Entity
girl = \ [x] -> elem x [A,C,D,G]
boy  = \ [x] -> elem x [B,E,F]
```

Some example binary relations:

```
love, hate :: Pred Entity
love = \ [x,y] -> elem (x,y) [(A,A),(A,B),(B,A),(C,B)]
hate = \ [x,y] -> elem (x,y) [(B,C),(C,D)]
```

And here is an example of a ternary relation:

```
give, introduce :: Pred Entity
give = \ [x,y,z] -> elem (x,y,z) [(A,H,B),(A,M,E)]
introduce = \ [x,y,z] -> elem (x,y,z) [(A,A,B),(A,B,C)]
```

The intention is that the first element in the list specifies the giver, the second element the receiver, and the third element what is given.

Operations on predications

Once we have this we can specify operations on predications. A simple example is passivization, which is a process of argument reduction: the agent of an action is dropped. Here is a possible implementation:

```
passivize :: [a] -> Pred a -> Pred a
passivize domain r = \ xs -> any (\ y -> r (y:xs)) domain
```

Let's check this out:

```
*IST> :t (passivize entities love)
(passivize entities love) :: Pred Entity
*IST> filter (\ x -> passivize entities love [x]) entities
[A,B]
```

Note that this also works for for ternary predicates. Here is the illustration:

```
*IST> :t (passivize entities give)
(passivize' entities give) :: Pred Entity
*IST> filter (passivize entities give)
         [[x,y] | x <- entities, y <- entities]
[[H,B],[M,E]]
```

Reflexivization

Another example of argument reduction in natural languages is reflexivization. The view that reflexive pronouns are relation reducers is folklore among logicians, but can also be found in linguistics textbooks, such as Daniel Büring's book on Binding Theory (Büring, 2005, pp. 43–45).

Under this view, reflexive pronouns like *himself* and *herself* differ semantically from non-reflexive pronouns like *him* and *her* in that they are not interpreted as individual variables. Instead, they denote argument reducing functions. Consider, for example, the following sentence:

Alice loved herself. (1)

The reflexive *herself* is interpreted as a function that takes the two-place predicate *loved* as an argument and turns it into a one-place predicate, which takes the subject as an argument, and expresses that this entity loves itself. This can be achieved by the following function `self`.

```
self ::  Pred a -> Pred a
self r = \ (x:xs) -> r (x:x:xs)
```

Here is an example application:

```
*IST> :t (self love)
(self love) :: Pred Entity
*IST> :t  \ x -> self love [x]
\ x -> self love [x] :: Entity -> Bool
*IST> filter (\ x -> self love [x]) entities
[A]
```

This approach to reflexives has two desirable consequences. The first one is that the locality of reflexives immediately falls out. Since `self` is applied to a predicate and unifies arguments of this predicate, it is not possible that an argument is unified with a non-clause mate. So in a sentence like (2), *herself* can only refer to *Alice* but not to *Carol*.

Carol believed that Alice loved herself. (2)

The second one is that it also immediately follows that reflexives in subject position are out.

**Herself loved Alice.* (3)

Given a compositional interpretation, we first apply the predicate *loved* to *Alice*, which gives us the one-place predicate $\lambda[x] \mapsto \text{love}\,[x, a]$. Then trying to apply the function `self` to this will fail, because it expects at least two arguments, and there is only one argument position left.

Reflexive pronouns can also be used to reduce ditransitive verbs to transitive verbs, in two possible ways: the reflexive can be the direct object or the indirect object:

Alice introduced herself to Bob. (4)

Bob gave the book to himself. (5)

The first of these is already taken care of by the reduction operation above. For the second one, here is an appropriate reduction function:

```
self' ::  Pred a -> Pred a
self' r = \ (x:y:xs) -> r (x:y:x:xs)
```

Quantifier scoping

Quantifier scope ambiguities can be dealt with in several ways. From the point of view of type theory it is attractive to view sequences of quantifiers as functions from relations to truth values. E.g., the sequence "every man, some woman" takes a binary relation $\lambda xy \cdot R[x, y]$ as input and yields *True* if and only if it is the case that for every man x there is some woman y for which $R[x, y]$ holds. To get the reversed scope reading, just swap the quantifier sequence, and transform the relation by swapping the first two argument places, as follows:

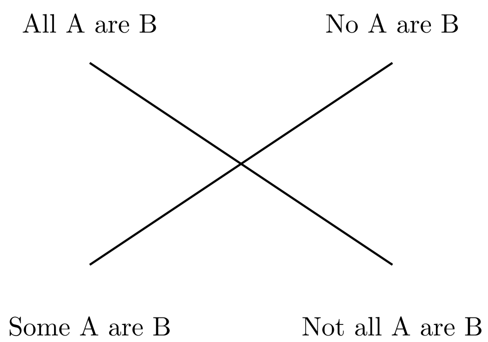
```
swap12 :: Pred a -> Pred a
swap12 r = \ (x:y:xs) -> r (y:x:xs)
```

So scope inversion can be viewed as a joint operation on quantifier sequences and relations. See (Eijck & Unger, 2010, Chapter 10) for a full-fledged implementation and for further discussion.

4 Example: Implementing Syllogistic Inference

As an example of the process of implementing inference for natural language, let us view the language of the Aristotelian syllogism as a tiny fragment of natural language. Compare the chapter by Larry Moss on Natural Logic in this Handbook. The treatment in this Section is an improved version of the implementation in (Eijck & Unger, 2010, Chapter 5).

The Aristotelian quantifiers are given in the following well-known square of opposition:



Aristotle interprets his quantifiers with existential import: *All A are B* and *No A are B* are taken to imply that there are A.

What can we ask or state with the Aristotelian quantifiers? The following grammar gives the structure of queries and statements (with PN for plural nouns):

$$
\begin{aligned}
Q ::= \;& \text{Are all PN PN?} \\
\mid\;& \text{Are no PN PN?} \\
\mid\;& \text{Are any PN PN?} \\
\mid\;& \text{Are any PN not PN?} \\
\mid\;& \text{What about PN?}
\end{aligned}
$$

$$
\begin{aligned}
S ::= \;& \text{All PN are PN.} \\
\mid\;& \text{No PN are PN.} \\
\mid\;& \text{Some PN are PN.} \\
\mid\;& \text{Some PN are not PN.}
\end{aligned}
$$

The meanings of the Aristotelean quantifiers can be given in terms of set inclusion and set intersection, as follows:

- **ALL**: Set inclusion
- **SOME**: Non-empty set intersection
- **NOT ALL**: Non-inclusion
- **NO**: Empty intersection

Set inclusion: $A \subseteq B$ holds if and only if every element of A is an element of B. Non-empty set intersection: $A \cap B \neq \emptyset$ if and only if there is some $x \in A$ with $x \in B$. Non-empty set intersection can can expressed in terms of inclusion, negation and complementation, as follows: $A \cap B \neq \emptyset$ if and only if $A \not\subseteq \overline{B}$.

To get a sound and complete inference system for this, we use the following **Key Fact:** A finite set of syllogistic forms Σ is unsatisfiable if and only if there exists an existential form ψ such that ψ taken together with the universal forms from Σ is unsatisfiable.

This restricted form of satisfiability can easily be tested with propositional logic. Suppose we talk about the properties of a single object x. Let proposition letter a express that object x has property A. Then a universal statement "All A are B" gets translated as $a \to b$. An existential statement "Some A is B" gets translated as $a \wedge b$.

For each property A we use a single proposition letter a. We have to check for *each* existential statement whether it is satisfiable when taken together with all universal statements. To test the satisfiability of a set of syllogistic statements with n existential statements we need n checks.

Literals, Clauses, Clause Sets

A *literal* is a propositional letter or its negation. A *clause* is a set of literals. A *clause set* is a set of clauses.

Read a clause as a *disjunction* of its literals, and a clause set as a *conjunction* of its clauses.

Represent the propositional formula

$$(p \to q) \wedge (q \to r)$$

as the following clause set:

$$\{\{\neg p, q\}, \{\neg q, r\}\}.$$

Here is an inference rule for clause sets: *unit propagation*

Unit Propagation

If one member of a clause set is a singleton $\{l\}$, then:

- remove every other clause containing l from the clause set;
- remove $\bar{l}$ from every clause in which it occurs.

The result of applying this rule is a simplified equivalent clause set. For example, unit propagation for $\{p\}$ to

$$\{\{p\}, \{\neg p, q\}, \{\neg q, r\}, \{p, s\}\}$$

yields

$$\{\{p\}, \{q\}, \{\neg q, r\}\}.$$

Applying unit propagation for $\{q\}$ to this result yields:

$$\{\{p\}, \{q\}, \{r\}\}.$$

The *Horn fragment* of propositional logic consists of all clause sets where every clause has *at most one positive literal*. Satisfiability for syllogistic forms containing exactly one existental statement translates to the Horn fragment of propositional logic. HORNSAT is the problem of testing Horn clause sets for satisfiability. Here is an algorithm for HORNSAT:

HORNSAT Algorithm

- If unit propagation yields a clause set in which units $\{l\}, \{\bar{l}\}$ occur, the original clause set is unsatisfiable.
- Otherwise the units in the result determine a satisfying valuation. Recipe: for all units $\{l\}$ occurring in the final clause set, map their proposition letter to the truth value that makes l true. Map all other proposition letters to false.

Here is an implementation. The definition of literals:

```
data Lit = Pos Name | Neg Name deriving Eq

instance Show Lit where
  show (Pos x) = x
  show (Neg x) = '-':x

neg :: Lit -> Lit
neg (Pos x) = Neg x
neg (Neg x) = Pos x
```

We can represent a clause as a list of literals:

```
type Clause = [Lit]
```

The names occurring in a list of clauses:

```
names :: [Clause] -> [Name]
names = sort . nub . map nm . concat
  where nm (Pos x) = x
        nm (Neg x) = x
```

The implementation of the unit propagation algorithm: propagation of a single unit literal:

```
unitProp :: Lit -> [Clause] -> [Clause]
unitProp x cs = concat (map (unitP x) cs)

unitP :: Lit -> Clause -> [Clause]
unitP x ys = if elem x ys then []
              else
               if elem (neg x) ys
                then [delete (neg x) ys]
                else [ys]
```

The property of being a unit clause:

```
unit :: Clause -> Bool
unit [x] = True
unit  _  = False
```

Propagation has the following type, where the Maybe expresses that the attempt to find a satisfying valuation may fail.

```
propagate :: [Clause] -> Maybe ([Lit],[Clause])
```

The implementation uses an auxiliary function `prop` with three arguments. The first argument gives the literals that are currently mapped to True, the second argument gives the literals that occur in unit clauses, the third argument gives the non-unit clauses.

```
propagate cls =
  prop [] (concat (filter unit cls)) (filter (not.unit) cls)
  where
    prop :: [Lit] -> [Lit] -> [Clause]
            -> Maybe ([Lit],[Clause])
    prop xs [] clauses = Just (xs,clauses)
    prop xs (y:ys) clauses =
      if elem (neg y) xs
       then Nothing
       else prop (y:xs)(ys++newlits) clauses' where
        newclauses = unitProp y clauses
        zs         = filter unit newclauses
        clauses'   = newclauses \\ zs
        newlits    = concat zs
```

Knowledge bases

A knowledge base is a pair, with as first element the clauses that represent the universal statements, and as second element a lists of clause lists, consisting of one clause list per existential statement.

```
type KB = ([Clause],[[Clause]])
```

The intention is that the first element represents the universal statements, while the second element has one clause list per existential statement.

The universe of a knowledge base is the list of all classes that are mentioned in it. We assume that classes are literals:

```
type Class = Lit

universe :: KB -> [Class]
universe (xs,yss) =
  map (\ x -> Pos x) zs ++ map (\ x -> Neg x) zs
    where zs = names (xs ++ concat yss)
```

Statements and queries according to the grammar given above:

```
data Statement =
     All1  Class   Class | No1     Class Class
   | Some1 Class   Class | SomeNot Class Class
   | AreAll Class Class  | AreNo   Class Class
   | AreAny Class Class  | AnyNot  Class Class
   | What   Class
  deriving Eq
```

A statement display function is given in the appendix. Statement classification:

```
isQuery :: Statement -> Bool
isQuery (AreAll _ _)  = True
isQuery (AreNo _ _)   = True
isQuery (AreAny _ _)  = True
isQuery (AnyNot _ _)  = True
isQuery (What _)      = True
isQuery  _            = False
```

Universal fact to statement. An implication $p \to q$ is represented as a clause $\{\neg p, q\}$, and yields a universal statement "All p are q". An implication $p \to \neg q$ is represented as a clause $\{\neg p, \neg q\}$, and yields a statement "No p are q".

```
u2s :: Clause -> Statement
u2s [Neg x, Pos y] = All1 (Pos x) (Pos y)
u2s [Neg x, Neg y] = No1 (Pos x) (Pos y)
```

Existential fact to statement. A conjunction $p \wedge q$ is represented as a clause set $\{\{p\}, \{q\}\}$, and yields an existential statement "Some p are q". A conjunction $p \wedge \neg q$ is represented as a clause set $\{\{p\}, \{\neg q\}\}$, and yields a statement "Some p are not q".

```
e2s :: [Clause] -> Statement
e2s [[Pos x],[Pos y]] = Some1 (Pos x) (Pos y)
e2s [[Pos x],[Neg y]] = SomeNot (Pos x) (Pos y)
```

Query negation:

```haskell
negat :: Statement -> Statement
negat (AreAll as bs) = AnyNot as bs
negat (AreNo  as bs) = AreAny as bs
negat (AreAny as bs) = AreNo  as bs
negat (AnyNot as bs) = AreAll as bs
```

The proper subset relation $\subset$ is computed as the list of all pairs (x, y) such that adding clauses $\{x\}$ and $\{\neg y\}$ — together these express that $x \cap \overline{y}$ is non-empty — to the universal statements in the knowledge base yields inconsistency.

```haskell
subsetRel :: KB -> [(Class,Class)]
subsetRel kb =
  [(x,y) | x <- classes, y <- classes,
     propagate ([x]:[neg y]: fst kb) == Nothing ]
    where classes = universe kb
```

If $R \subseteq A^2$ and $x \in A$, then $xR := \{y \mid (x, y) \in R\}$. This is called a *right section of a relation.*

```haskell
rSection :: Eq a => a -> [(a,a)] -> [a]
rSection x r = [ y | (z,y) <- r, x == z ]
```

The supersets of a class are given by a right section of the subset relation, that is, the supersets of a class are all classes of which it is a subset.

```haskell
supersets :: Class -> KB -> [Class]
supersets cl kb = rSection cl (subsetRel kb)
```

The non-empty intersection relation is computed by combining each of the existential clause lists form the knowledge base with the universal clause list.

```haskell
intersectRel :: KB -> [(Class,Class)]
intersectRel kb@(xs,yys) =
 nub [(x,y) | x <- classes, y <- classes, lits <- litsList,
    elem x lits && elem y lits   ]
     where
       classes = universe kb
       litsList =
         [ maybe [] fst (propagate (ys++xs)) | ys <- yys ]
```

The intersection sets of a class C are the classes that have a non-empty intersection with C:

```
intersectionsets :: Class -> KB -> [Class]
intersectionsets cl kb = rSection cl (intersectRel kb)
```

In general, in KB query, there are three possibilities:

(1) `derive kb stmt` is true. This means that the statement is derivable, hence true.
(2) `derive kb (neg stmt)` is true. This means that the negation of `stmt` is derivable, hence true. So `stmt` is false.
(3) neither `derive kb stmt` nor `derive kb (neg stmt)` is true. This means that the knowledge base has no information about `stmt`.

The derivability relation is given by:

```
derive :: KB -> Statement -> Bool
derive kb (AreAll as bs) = bs `elem` (supersets as kb)
derive kb (AreNo as bs)  = (neg bs) `elem` (supersets as kb)
derive kb (AreAny as bs) = bs `elem` (intersectionsets as kb)
derive kb (AnyNot as bs) = (neg bs) `elem`
                              (intersectionsets as kb)
```

To build a knowledge base we need a function for updating an existing knowledge base with a statement. If the update is successful, we want an updated knowledge base. If the update is not successful, we want to get an indication of failure. This explains the following type. The boolean in the output is a flag indicating change in the knowledge base.

```
update  :: Statement -> KB -> Maybe (KB,Bool)
```

Update with an ‘All’ statement. The update function checks for possible inconsistencies. E.g., a request to add an $A \subseteq B$ fact to the knowledge base leads to an inconsistency if $A \not\subseteq B$ is already derivable.

```
update (All1 as bs) kb@(xs,yss)
  | bs' `elem` (intersectionsets as kb) = Nothing
  | bs `elem` (supersets  as kb) = Just (kb,False)
  | otherwise = Just (([as',bs]:xs,yss),True)
 where
   as' = neg as
   bs' = neg bs
```

Update with other kinds of statements:

```
update (No1 as bs) kb@(xs,yss)
  | bs `elem` (intersectionsets as kb) = Nothing
  | bs' `elem` (supersets  as kb) = Just (kb,False)
  | otherwise = Just (([as',bs']:xs,yss),True)
 where
   as' = neg as
   bs' = neg bs
```

```
update (Some1 as bs) kb@(xs,yss)
  | bs' `elem` (supersets  as kb) = Nothing
  | bs `elem` (intersectionsets as kb) = Just (kb,False)
  | otherwise = Just ((xs,[[as],[bs]]:yss),True)
 where
   bs' = neg bs
```

```
update (SomeNot as bs) kb@(xs,yss)
  | bs `elem` (supersets  as kb) = Nothing
  | bs' `elem` (intersectionsets as kb) = Just (kb,False)
  | otherwise = Just ((xs,[[as],[bs']]:yss),True)
 where
   bs' = neg bs
```

The above implementation of an inference engine for syllogistic reasoning is a mini-case of computational semantics. What is the use of this? Cognitive research focusses on this kind of quantifier reasoning, so it is a pertinent question whether the engine can be used to meet cognitive realities? A possible link with cognition would refine this calculus and the check whether the predictions for differences in processing speed for various tasks are realistic.

There is also a link to the "natural logic for natural language" enterprise: the logical forms for syllogistic reasoning are very close to the surface forms of the sentences. The Chapter on Natural Logic in this Handbook gives more information. All in all, reasoning engines like this one are relevant for rational reconstructions of cognitive processing. The appendix gives the code for constructing a knowledge base from a list of statements, and updating it. Here is a chat function that starts an interaction from a given knowledge base and writes the result of the interaction to a file:

```
chat :: IO ()
chat = do
 kb <- getKB "kb.txt"
 writeKB "kb.bak" kb
 putStrLn "Update or query the KB:"
 str <- getLine
 if str == "" then return ()
  else do
   handleCases kb str
   chat
```

You are invited to try this out by loading the software for this chapter and running `chat`.

5 Implementing Fragments of Natural Language

Now what about the meanings of the sentences in a simple fragment of English? Using what we know now about a logical form language and its interpretation in appropriate models, and assuming we have constants available for proper names, and predicate letters for the nouns and verbs of the fragment, we can easily translate the sentences generated by a simple example grammar into logical forms. Assume the following translation key:

lexical item	translation	type of logical constant
girl	*Girl*	one-place predicate
boy	*Boy*	one-place predicate
toy	*Toy*	one-place predicate
laughed	*Laugh*	one-place predicate
cheered	*Cheer*	one-place predicate
loved	*Love*	two-place predicate
admired	*Admire*	two-place predicate
helped	*Help*	two-place predicate
defeated	*Defeat*	two-place predicate
gave	*Give*	three-place predicate
introduced	*Introduce*	three-place predicate
Alice	a	individual constant
Bob	b	individual constant
Carol	c	individual constant

Then the translation of *Every boy loved a girl* in the logical form language above could become:

$$Q_{\forall}x(\mathit{Boy}\ x)(Q_{\exists}y(\mathit{Girl}\ y)(\mathit{Love}\ x\ y)).$$

To start the construction of meaning representations, we first represent a context free grammar for a natural language fragment in Haskell. A rule like $S ::= NP\ VP$ defines syntax trees consisting of an S node immediately dominating an NP node and a VP node. This is rendered in Haskell as the following datatype definition:

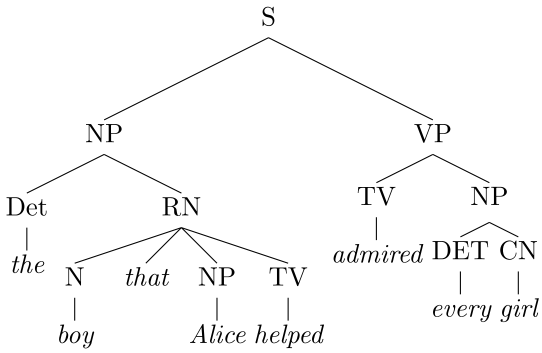
```
data S = S NP VP
```

The `S` on the righthand side is a combinator indicating the name of the top of the tree. Here is a grammar for a tiny fragment:

```
data S = S NP VP deriving Show
data NP = NP1 NAME | NP2 Det N | NP3 Det RN
  deriving Show
data ADJ = Beautiful | Happy | Evil
  deriving Show
data NAME = Alice | Bob | Carol
  deriving Show
data N = Boy | Girl | Toy | N ADJ N
  deriving Show
data RN = RN1 N That VP | RN2 N That NP TV
  deriving Show
data That = That deriving Show
data VP = VP1 IV | VP2 TV NP | VP3 DV NP NP deriving Show
data IV = Cheered | Laughed deriving Show
data TV = Admired | Loved | Hated | Helped deriving Show
data DV = Gave | Introduced deriving Show
```

Look at this as a definition of syntactic structure trees. The structure for *The boy that Alice helped admired every girl* is given in Figure 1, with the Haskell version of the tree below it.

Figure 1. Example structure tree




```
S
 (NP (Det the)
     (RN (N boy) That  (NP Alice) (TV helped))
 (VP  (TV admired) (NP (DET every) (N girl)))
```

For the purpose of this chapter we skip the definition of the parse function that maps the string *The boy that Alice helped admired every girl* to this structure (but see (Eijck & Unger, 2010, Chapter 9)).

Now all we have to do is find appropriate translations for the categories in the grammar of the fragment. The first rule, **S** $\longrightarrow$ **NP VP**, already presents us with a difficulty. In looking for NP translations and VP translations, should we represent NP as a function that takes a VP representation as argument, or vice versa?

In any case, VP representations will have a functional type, for VPs denote properties. A reasonable type for the function that represents a VP is `Term -> Formula`. If we feed it with a term, it will yield a logical form. Proper names now can get the type of terms. Take the example *Alice laughed*. The verb *laughed* gets represented as the function that maps the term `x` to the formula `Atom "laugh" [x]`. Therefore, we get an appropriate logical form for the sentence if `x` is a term for *Alice*.

A difficulty with this approach is that phrases like *no boy* and *every girl* do not fit into this pattern. Following Montague, we can solve this by assuming that such phrases translate into functions that take VP representations as arguments. So the general pattern becomes: the NP representation is the function that takes the VP representation as its argument. This gives:

```
lfS :: S -> Formula
lfS (S np vp) = (lfNP np) (lfVP vp)
```

Next, NP-representations are of type `(Term -> Formula) -> Formula`.

```
lfNP :: NP -> (Term -> Formula) -> Formula
lfNP (NP1 Alice)   = \ p -> p (Struct "Alice" [])
lfNP (NP1 Bob)     = \ p -> p (Struct "Bob"   [])
lfNP (NP1 Carol)   = \ p -> p (Struct "Carol" [])
lfNP (NP2 det cn)  = (lfDET det) (lfN cn)
lfNP (NP3 det rcn) = (lfDET det) (lfRN rcn)
```

Verb phrase representations are of type `Term -> Formula`.

```
lfVP :: VP -> Term -> Formula
lfVP (VP1 Laughed)   = \ t -> Atom "laugh"   [t]
lfVP (VP1 Cheered)   = \ t -> Atom "cheer"   [t]
```

Representing a function that takes two arguments can be done either by means of `a -> a -> b` or by means of `(a,a) -> b`. A function of the first type is called *curried*, a function of the second type *uncurried*.

We assume that representations of transitive verbs are uncurried, so they have type `(Term,Term) -> Formula`, where the first term slot is for the subject, and the second term slot for the object. Accordingly, the representations of ditransitive verbs have type

```
(Term,Term,Term) -> Formula
```

where the first term slot is for the subject, the second one is for the indirect object, and the third one is for the direct object. The result should in both cases be a property for VP subjects. This gives us:

```
lfVP (VP2 tv np) =
    \ subj -> lfNP np (\ obj -> lfTV tv (subj,obj))
lfVP (VP3 dv np1 np2) =
    \ subj -> lfNP np1 (\ iobj -> lfNP np2 (\ dobj ->
                          lfDV dv (subj,iobj,dobj)))
```

Representations for transitive verbs are:

```
lfTV :: TV -> (Term,Term) -> Formula
lfTV Admired  = \ (t1,t2) -> Atom "admire" [t1,t2]
lfTV Hated    = \ (t1,t2) -> Atom "hate" [t1,t2]
lfTV Helped   = \ (t1,t2) -> Atom "help" [t1,t2]
lfTV Loved    = \ (t1,t2) -> Atom "love" [t1,t2]
```

Ditransitive verbs:

```
lfDV :: DV -> (Term,Term,Term) -> Formula
lfDV Gave = \ (t1,t2,t3) -> Atom "give" [t1,t2,t3]
lfDV Introduced = \ (t1,t2,t3) ->
                    Atom "introduce" [t1,t2,t3]
```

Common nouns have the same type as VPs.

```
lfN :: N -> Term -> Formula
lfN Girl     = \ t -> Atom "girl"     [t]
lfN Boy      = \ t -> Atom "boy"      [t]
```

The determiners we have already treated above, in Section 2. Complex common nouns have the same types as simple common nouns:

```
lfRN :: RN -> Term -> Formula
lfRN (RN1 cn _ vp)    = \ t -> Cnj [lfN cn t, lfVP vp t]
lfRN (RN2 cn _ np tv) = \ t -> Cnj [lfN cn t,
                         lfNP np (\ subj -> lfTV tv (subj,t))]
```

We end with some examples:

```
lf1 = lfS (S (NP2 Some Boy)
                   (VP2 Loved (NP2 Some Girl)))
lf2 = lfS (S (NP3 No (RN2 Girl That (NP1 Bob) Loved))
                   (VP1 Laughed))
lf3 = lfS (S (NP3 Some (RN1 Girl That (VP2 Helped (NP1 Alice))))
                   (VP1 Cheered))
```

This gives:

```
*IST> lf1
Q Some x2 (Atom "boy" [x2])
          (Q Some x1 (Atom "girl" [x1]) (Atom "love" [x2,x1]))
*IST> lf2
Q No x1 (Cnj [Atom "girl" [x1],Atom "love" [Bob,x1]])
        (Atom "laugh" [x1])
*IST> lf3
Q Some x1 (Cnj [Atom "girl" [x1],Atom "help" [x1,Alice]])
         (Atom "cheer" [x1])
```

What we have presented here is in fact an implementation of an extensional fragment of Montague grammar. The next Section indicates what has to change in an intensional fragment.

6 Extension and Intension

One of the trademarks of Montague grammar is the use of possible worlds to treat intensionality. Instead of giving a predicate a single interpretation in a model, possible world semantics gives intensional predicates different interpretations in different situations (or: in different "possible worlds"). A prince in one world may be a beggar in another, and the way in which intensional semantics accounts for this is by giving predicates like *prince* and *beggar* different interpretations in different worlds.

So we assume that apart from entities and truth values there is another basic type, for possible worlds. We introduce names or indices for possible worlds, as follows:

```
data World = W Int deriving (Eq,Show)
```

Now the type of *individual concepts* is the type of functions from worlds to entities, i.e., `World -> Entity`. An individual concept is a *rigid designator* if it picks the same entity in every possible world:

```
rigid :: Entity -> World -> Entity
rigid x = \ _ -> x
```

A function from possible worlds to truth values is a *proposition*. Propositions have type `World -> Bool`. In *Mary desires to marry a prince* the rigid designator that interprets the proper name "Mary" is related to a proposition, namely the proposition that is true in a world if and only if Mary marries someone who, in that world, is a prince. So an intensional verb like *desire* may have type `(World -> Bool) -> (World -> Entity) -> Bool`, where `(World -> Bool)` is the type of "marry a prince", and `(World -> Entity)` is the type for the intensional function that interprets "Mary."

Models for intensional logic have a domain D of entities plus functions from predicate symbols to intensions of relations. Here is an example interpretion for the predicate symbol "princess:"

```
princess :: World -> Pred Entity
princess = \ w [x] -> case w of
             W 1 -> elem x [A,C,D,G]
             W 2 -> elem x [A,M]
             _   -> False
```

What this says is that in W_1 x is a princess iff x is among A, C, D, G, in W_2 x is a princess iff x is among A, M, and in no other world is x a princess. This interpretation for "princess" will make "Mary is a princess" true in W_2 but in no other world.

7 Implementing Communicative Action

The simplest kind of communicative action probably is *question answering* of the kind that was demonstrated in the Syllogistics tool above, in Section 4. The interaction is between a system (the knowledge base) and a user. In the implementation we only keep track of changes in the system: the knowledge base gets updated every time the user makes statements that are consistent with the knowledge base but not derivable from it.

Generalizing this, we can picture a group of communicating agents, each with their own knowledge, with acts of communication that change these knowledge bases. The basic logical tool for this is again intensional logic, more in particular the epistemic logic proposed by Hintikka in Hintikka (1962), and adapted in cognitive science (Gärdenfors (1988)), computer science (Fagin *et al.* (1995)) and economics (Aumann (1976); Battigalli & Bonanno (1999)). The general system for tracking how knowledge and belief of communicating agents evolve under various kinds of communication is called *dynamic epistemic logic* or *DEL*. See van Benthem (2011) for a general perspective, and Ditmarsch *et al.* (2006) for a textbook account.

To illustrate the basics, we will give an implementation of model checking for epistemic update logic with public announcements.

The basic concept in the logic of knowledge is that of epistemic uncertainty. If I am uncertain about whether a coin that has just been tossed is showing head or tail, this can be pictured as two situations related by my uncertainty. Such uncertainty relations are equivalences: If I am uncertain between situations s and t, and between situations t and r, this means I am also uncertain between s and r.

Equivalence relations on a set of situations S can be implemented as partitions of S, where a partition is a family X_i of sets with the following properties (let I be the index set):

- For each $i \in I$, $X_i \neq \emptyset$ and $X_i \subseteq S$.
- For $i \neq j$, $X_i \cap X_j = \emptyset$.
- $\bigcup_{i \in I} X_i = S$.

Here is a datatype for equivalence relations, viewed as partitions (lists of lists of items):

```
type Erel a = [[a]]
```

The block of an item x in a partition is the set of elements that are equivalent to x:

```
bl :: Eq a => Erel a -> a -> [a]
bl r x = head (filter (elem x) r)
```

The restriction of a partition to a domain:

```
restrict :: Eq a => [a] -> Erel a -> Erel a
restrict domain =  nub . filter (/= [])
                   . map (filter (flip elem domain))
```

An infinite number of agents, with names a, b, c, d, e for the first five of them:

```
data Agent = Ag Int deriving (Eq,Ord)

a,b,c,d,e :: Agent
a = Ag 0; b = Ag 1; c = Ag 2; d = Ag 3; e = Ag 4

instance Show Agent where
  show (Ag 0) = "a"; show (Ag 1) = "b"; show (Ag 2) = "c";
  show (Ag 3) = "d"; show (Ag 4) = "e";
  show (Ag n) = 'a': show n
```

A datatype for epistemic models:

```
data EpistM state = Mo
             [state]
             [Agent]
             [(Agent,Erel state)]
             [state]  deriving (Eq,Show)
```

An example epistemic model:

```
example :: EpistM Int
example = Mo
 [0..3]
 [a,b,c]
 [(a,[[0],[1],[2],[3]]),(b,[[0],[1],[2],[3]]),(c,[[0..3]])]
 [1]
```

In this model there are three agents and four possible worlds. The first two agents a and b can distinguish all worlds, and the third agent c confuses all of them.

Extracting an epistemic relation from a model:

```
rel :: Agent -> EpistM a -> Erel a
rel ag (Mo _ _ rels _) = myLookup ag rels

myLookup :: Eq a => a -> [(a,b)] -> b
myLookup x table =
   maybe (error "item not found") id (lookup x table)
```

This gives:

```
*IST> rel a example
[[0],[1],[2],[3]]
*IST> rel c example
[[0,1,2,3]]
*IST> rel d example
*** Exception: item not found
```

A logical form language for epistemic statements; note that the type has a parameter for additional information.

```
data Form a = Top
            | Info a
            | Ng (Form a)
            | Conj [Form a]
            | Disj [Form a]
            | Kn Agent (Form a)
          deriving (Eq,Ord,Show)
```

A useful abbreviation:

```
impl :: Form a -> Form a -> Form a
impl form1 form2 = Disj [Ng form1, form2]
```

Semantic interpretation for this logical form language:

```
isTrueAt :: Ord state =>
            EpistM state -> state -> Form state -> Bool
isTrueAt m w Top = True
isTrueAt m w (Info x) = w == x
isTrueAt m w (Ng f) = not (isTrueAt m w f)
isTrueAt m w (Conj fs) = and (map (isTrueAt m w) fs)
isTrueAt m w (Disj fs) = or  (map (isTrueAt m w) fs)
isTrueAt
 m@(Mo worlds agents acc points) w (Kn ag f) = let
    r = rel ag m
    b = bl r w
  in
    and (map (flip (isTrueAt m) f) b)
```

This treats the Boolean connectives as usual, and interprets knowledge as truth in all worlds in the current accessible equivalence block of an agent.

The effect of a public announcement ϕ on an epistemic model is that the set of worlds of that model gets limited to the worlds where ϕ is true, and the accessibility relations get restricted accordingly.

```
upd_pa :: Ord state =>
          EpistM state -> Form state -> EpistM state
upd_pa m@(Mo states agents rels actual) f =
  (Mo states' agents rels' actual')
   where
   states' = [ s | s <- states, isTrueAt m s f ]
   rels'   = [(ag,restrict states' r) | (ag,r) <- rels ]
   actual' = [ s | s <- actual, s `elem` states' ]
```

A series of public announcement updates:

```
upds_pa ::  Ord state =>
            EpistM state -> [Form state] -> EpistM state
upds_pa m [] = m
upds_pa m (f:fs) = upds_pa (upd_pa m f) fs
```

We illustrate the working of the update mechanism on a famous epistemic puzzle. The following Sum and Product riddle was stated by the Dutch mathematican Hans Freudenthal in a Dutch mathematics journal in 1969. There is also a version by John McCarthy (see `http://www-formal.stanford.edu/jmc/puzzles.htm`).

> A says to S and P: I have chosen two integers x, y such that $1 < x < y$ and $x + y \leq 100$. In a moment, I will inform S only of $s = x + y$, and

> P only of $p = xy$. These announcements remain private. You are required to determine the pair (x, y). He acts as said. The following conversation now takes place:
> (1) P says: "I do not know the pair."
> (2) S says: "I knew you didn't."
> (3) P says: "I now know it."
> (4) S says: "I now also know it."
> Determine the pair (x, y).

This was solved by combinatorial means in a later issue of the journal. A model checking solution with DEMO Eijck (2007) (based on a DEMO program written by Ji Ruan) was presented in Ditmarsch *et al.* (2005). The present program is an optimized version of that solution.

The list of candidate pairs:

```
pairs :: [(Int,Int)]
pairs  = [ (x,y) |  x <- [2..100], y <- [2..100],
                    x < y, x+y <= 100 ]
```

The initial epistemic model is such that a (representing S) cannot distinguish number pairs with the same sum, and b (representing P) cannot distinguish number pairs with the same product. Instead of using a valuation, we use number pairs as worlds.

```
msnp :: EpistM (Int,Int)
msnp = (Mo pairs [a,b] acc pairs)
  where
  acc   = [ (a, [ [ (x1,y1) | (x1,y1) <- pairs,
                              x1+y1 == x2+y2 ] |
                                (x2,y2) <- pairs ] ) ]
          ++
          [ (b, [ [ (x1,y1) | (x1,y1) <- pairs,
                              x1*y1 == x2*y2 ] |
                                (x2,y2) <- pairs ] ) ]
```

The statement by b that he does not know the pair:

```
statement_1 =
  Conj [ Ng (Kn b (Info p)) | p <- pairs ]
```

To check this statement is expensive. A computationally cheaper equivalent statement is the following (see Ditmarsch *et al.* (2005)).

```
statement_1e =
  Conj [ Info p `impl` Ng (Kn b (Info p)) | p <- pairs ]
```

In Freudenthal's story, the first public announcement is the statement where b confesses his ignorance, and the second public announcement is the statement by a about her knowledge about b's state of knowledge *before* that confession. We can wrap the two together in a single statement to the effect that initially, a knows that b does not know the pair. This gives:

```
k_a_statement_1e = Kn a statement_1e
```

The second announcement proclaims the statement by b that now he knows:

```
statement_2 =
  Disj [ Kn b (Info p) | p <- pairs ]
```

Equivalently, but computationally more efficient:

```
statement_2e =
  Conj [ Info p `impl` Kn b (Info p) | p <- pairs ]
```

The final announcement concerns the statement by a that now she knows as well.

```
statement_3 =
  Disj [ Kn a (Info p) | p <- pairs ]
```

In the computationally optimized version:

```
statement_3e =
  Conj [ Info p `impl` Kn a (Info p) | p <- pairs ]
```

The solution:

```
solution = upds_pa msnp
             [k_a_statement_1e,statement_2e,statement_3e]
```

This is checked in a matter of minutes:

```
*IST> solution
Mo [(4,13)] [a,b] [(a,[[(4,13)]]),(b,[[(4,13)]])] [(4,13)]
```

8 Resources

Code for this Chapter

The example code in this Chapter can be found at internet address `https://github.com/janvaneijck/ist`. To run this software, you will need the Haskell system, which can be downloaded from `www.haskell.org`. This site also gives many interesting Haskell resources.

Epistemic model checking

More information on epistemic model checking can be found in the documentation of the epistemic model checker DEMO. See Eijck (2007).

Link for Computational Semantics With Functional Programming

The book Eijck & Unger (2010) has a website devoted to it, which can be found at `www.computationalsemantics.eu`.

Further computational semantics links

Special Interest Group in Computational Semantics: `http://www.sigsem.org/wiki/`. International Workshop on Computational Semantics: `http://iwcs.uvt.nl/`. Wikipedia entry on computational semantics: `http://en.wikipedia.org/wiki/Computational_semantics`.

9 Appendix

A show function for identifiers:

```
instance Show Id where
  show (Id name 0) = name
  show (Id name i) = name ++ show i
```

A show function for terms:

```
instance Show Term where
  show (Var id)   = show id
  show (Struct name []) = name
  show (Struct name ts) = name ++ show ts
```

For the definition of fresh variables, we collect the list of indices that are used in the formulas in the scope of a quantifier, and select a fresh index, i.e., an index that does not occur in the index list:

```
fresh :: [Formula] -> Int
fresh fs = i+1 where i = maximum (0:indices fs)

indices :: [Formula] -> [Int]
indices [] = []
indices (Atom _ _:fs) = indices fs
indices (Eq _ _:fs) = indices fs
indices (Not f:fs)  = indices (f:fs)
indices (Cnj fs1:fs2) = indices (fs1 ++ fs2)
indices (Dsj fs1:fs2) = indices (fs1 ++ fs2)
indices (Q _ (Id _ n) f1 f2:fs) = n : indices (f1:f2:fs)
```

A show function for the statements in our syllogistic inference fragment:

964

```
instance Show Statement where
  show (All1 as bs)     =
    "All "  ++ show as ++ " are " ++ show bs ++ "."
  show (No1 as bs)      =
    "No "   ++ show as ++ " are " ++ show bs ++ "."
  show (Some1 as bs)    =
    "Some " ++ show as ++ " are " ++ show bs ++ "."
  show (SomeNot as bs) =
    "Some " ++ show as ++ " are not " ++ show bs ++ "."
  show (AreAll as bs)  =
    "Are all " ++ show as ++ show bs ++ "?"
  show (AreNo as bs)   =
    "Are no "  ++ show as ++ show bs ++ "?"
  show (AreAny as bs)  =
    "Are any " ++ show as ++ show bs ++ "?"
  show (AnyNot as bs)  =
    "Are any " ++ show as ++ " not " ++ show bs ++ "?"
  show (What as)       = "What about " ++ show as ++ "?"
```

965 Constructing a knowledge base from a list of statements:

966

```
makeKB :: [Statement] -> Maybe KB
makeKB = makeKB' ([],[])
     where
         makeKB' kb []     = Just kb
         makeKB' kb (s:ss) = case update s kb of
              Just (kb',_) -> makeKB' kb' ss
              Nothing      -> Nothing
```

967 A preprocess function to prepare for parsing:

968

```
preprocess :: String -> [String]
preprocess = words . (map toLower) .
             (takeWhile (\ x -> isAlpha x || isSpace x))
```

969 A parse function, with a type indicating that the parsing may fail:

```
parse :: String -> Maybe Statement
parse = parse' . preprocess
  where
    parse' ["all",as,"are",bs] =
      Just (All1 (Pos as) (Pos bs))
    parse' ["no",as,"are",bs] =
      Just (No1 (Pos as) (Pos bs))
    parse' ["some",as,"are",bs] =
      Just (Some1 (Pos as) (Pos bs))
    parse' ["some",as,"are","not",bs] =
      Just (SomeNot (Pos as) (Pos bs))
970 parse' ["are","all",as,bs] =
      Just (AreAll (Pos as) (Pos bs))
    parse' ["are","no",as,bs] =
      Just (AreNo (Pos as) (Pos bs))
    parse' ["are","any",as,bs] =
      Just (AreAny (Pos as) (Pos bs))
    parse' ["are","any",as,"not",bs] =
      Just (AnyNot (Pos as) (Pos bs))
    parse' ["what", "about", as] = Just (What (Pos as))
    parse' ["how", "about", as]  = Just (What (Pos as))
    parse' _ = Nothing
```

971 Processing a piece of text, given as a string with newline characters.

```
process :: String -> KB
972 process txt =
  maybe ([],[]) id (mapM parse (lines txt) >>= makeKB)
```

973 An example text, consisting of lines separated by newline characters:

```
mytxt = "all bears are mammals\n"
     ++ "no owls are mammals\n"
     ++ "some bears are stupids\n"
     ++ "all men are humans\n"
974  ++ "no men are women\n"
     ++ "all women are humans\n"
     ++ "all humans are mammals\n"
     ++ "some men are stupids\n"
     ++ "some men are not stupids"
```

975 Reading a knowledge base from disk:

976

```
getKB :: FilePath -> IO KB
getKB p = do
           txt <- readFile p
           return (process txt)
```

977 Writing a knowledge base to disk, in the form of a list of statements.

978

```
writeKB :: FilePath -> KB -> IO ()
writeKB p (xs,yss) = writeFile p (unlines (univ ++ exist))
  where
    univ  = map (show.u2s) xs
    exist = map (show.e2s) yss
```

979 Telling about a class, based on the info in a knowledge base.

980

```
tellAbout :: KB -> Class -> [Statement]
tellAbout kb as =
  [All1 as (Pos bs) | (Pos bs) <- supersets as kb,
                       as /= (Pos bs) ]
  ++
  [No1  as (Pos bs) | (Neg bs) <- supersets as kb,
                       as /= (Neg bs) ]
  ++
  [Some1 as (Pos bs) | (Pos bs) <- intersectionsets as kb,
                     as /= (Pos bs),
                     notElem (as,Pos bs) (subsetRel kb) ]
  ++
  [SomeNot as (Pos bs) | (Neg bs) <- intersectionsets as kb,
                   notElem (as, Neg bs) (subsetRel kb) ]
```

981 Depending on the input, the various cases are handled by the following
982 function:

```
handleCases :: KB -> String -> IO ()
handleCases kb str =
   case parse str of
     Nothing        -> putStrLn "Wrong input.\n"
     Just (What as) -> let
         info = (tellAbout kb as, tellAbout kb (neg as)) in
       case info of
        ([],[])      -> putStrLn "No info.\n"
        ([],negi)    -> putStrLn (unlines (map show negi))
        (posi,negi)  -> putStrLn (unlines (map show posi))
     Just stmt      ->
      if isQuery stmt then
        if derive kb stmt then putStrLn "Yes.\n"
          else if derive kb (negat stmt)
                 then putStrLn "No.\n"
                 else putStrLn "I don’t know.\n"
        else case update stmt kb of
          Just (kb’,True) -> do
                              writeKB "kb.txt" kb’
                              putStrLn "OK.\n"
          Just (_,False)  -> putStrLn
                              "I knew that already.\n"
          Nothing         -> putStrLn
                              "Inconsistent with my info.\n"
```

References

Alshawi, H. (ed.) (1992), *The Core Language Engine*, MIT Press, Cambridge Mass, Cambridge, Mass., and London, England.

Alshawi, H. & J. van Eijck (1989), Logical forms in the core language engine, in *Proceedings of the 27th Congress of the ACL*, ACL, Vancouver.

Aumann, R.J. (1976), Agreeing to disagree, *Annals of Statistics* 4(6):1236–1239.

Barwise, J. & R. Cooper (1981), Generalized quantifiers and natural language, *Linguistics and Philosophy* 4:159–219.

Battigalli, P. & G. Bonanno (1999), Recent results on belief, knowledge and the epistemic foundations of game theory, *Research in Economics* 53:149–225.

van Benthem, J. (2011), *Logical Dynamics of Information and Interaction*, Cambridge University Press.

Blackburn, P. & J. Bos (2005), *Representation and Inference for Natural Language; A First Course in Computational Semantics*, CSLI Lecture Notes.

Büring, D. (2005), *Binding Theory*, Cambridge Textbooks in Linguistics, Cambridge University Press.

Ditmarsch, Hans van, Ji Ruan, & Rineke Verbrugge (2005), Model checking sum and product, in Shichao Zhang & Ray Jarvis (eds.), *AI 2005: Advances in Artificial Intelligence: 18th Australian Joint Conference on Artificial Intelligence*, Springer-Verlag GmbH, volume 3809 of *Lecture Notes in Computer Science*, (790–795).

Ditmarsch, H.P. van, W. van der Hoek, & B. Kooi (2006), *Dynamic Epistemic Logic*, volume 337 of *Synthese Library*, Springer.

Eijck, Jan van (2007), DEMO — a demo of epistemic modelling, in Johan van Benthem, Dov Gabbay, & Benedikt Löwe (eds.), *Interactive Logic — Proceedings of the 7th Augustus de Morgan Workshop*, Amsterdam University Press, number 1 in Texts in Logic and Games, (305–363).

Eijck, Jan van & Christina Unger (2010), *Computational Semantics with Functional Programming*, Cambridge University Press.

Fagin, R., J.Y. Halpern, Y. Moses, & M.Y. Vardi (1995), *Reasoning about Knowledge*, MIT Press.

Gärdenfors, P. (1988), *Knowledge in Flux: Modelling the Dynamics of Epistemic States*, MIT Press, Cambridge Mass.

Hintikka, J. (1962), *Knowledge and Belief: An Introduction to the Logic of the Two Notions*, Cornell University Press, Ithaca N.Y.

Hughes, J. (1989), Why functional programming matters, *The Computer Journal* 32(2):98–107, ISSN 0010-4620, doi:10.1093/comjnl/32.2.98.

Knuth, D.E. (1992), *Literate Programming*, CSLI Lecture Notes, no. 27, CSLI, Stanford.

Montague, R. (1973), The proper treatment of quantification in ordinary English, in J. Hintikka (ed.), *Approaches to Natural Language*, Reidel, (221–242).

Montague, R. (1974a), English as a formal language, in R.H. Thomason (ed.), *Formal Philosophy; Selected Papers of Richard Montague*, Yale University Press, New Haven and London, (188–221).

Montague, R. (1974b), Universal grammar, in R.H. Thomason (ed.), *Formal Philosophy; Selected Papers of Richard Montague*, Yale University Press, New Haven and London, (222–246).

Tarski, A. (1956), The concept of truth in the languages of the deductive sciences, in J. Woodger (ed.), *Logic, Semantics, Metamathematics*, Oxford, first published in Polish in 1933.

Index