

1

2 Implementing Semantic Theories

3

Jan van Eijck¹

4

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

6

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

45 To be suitable for the representation of semantic theories, an implemen-
 46 tation language has to have good facilities for specifying *abstract data types*.
 47 The key feature in specifying abstract data types is to present a precise de-
 48 scription of that data type without referring to any concrete representation
 49 of the objects of that datatype and to specify operations on the data type
 50 without referring to any implementation details.

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

56 *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 *Linear* resolution with a *Selection* function for
Definite 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:

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

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

62 Pure lambda calculus was developed in the 1930s and 40s by the logician
 63 Alonzo Church, as a foundational project intended to put mathematics on
 64 a firm basis of ‘effective procedures’. In the system of pure lambda calculus,
 65 *everything* is a function. Functions can be applied to other functions to obtain
 66 values by a process of application, and new functions can be constructed from
 67 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 (essen-
 tially procedures for separating the members of a set from the non-members),
 we can define

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

68 Now apply *r* to itself:

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

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

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

85 A declarative programming language is better than an imperative pro-
86 gramming language for implementing a description of a set of semantic rules.
87 The two main declarative programming styles that are considered suitable for
88 implementating computational semantics are logic programming and func-
89 tional programming. Indeed, computational paradigms that emerged in com-
90 puter science, such as unification and proof search, found their way into seman-
91 tic theory, as basic feature value computation mechanisms and as resolution
92 algorithms for pronoun reference resolution.

93 If unification and first order inference play an important role in a semantic
94 theory, then a logic programming language like Prolog may seem a natural
95 choice as an implementation language. However, while unification and proof
96 search for definite clauses constitute the core of logic programming (there is
97 hardly more to Prolog than these two ingredients), functional programming
98 encompasses the whole world of abstract datatype definition and polymorphic
99 typing. As we will demonstrate below, the key ingredients of logic program-
100 ming are easily expressed in Haskell, while Prolog is not very suitable for
101 expressing data abstraction. Therefore, in this chapter we will use Haskell
102 rather than Prolog as our implementation language. For a textbook on com-
103 putational semantics that uses Prolog, we refer to Blackburn & Bos (2005). A
104 recent computational semantics textbook that uses Haskell is Eijck & Unger
105 (2010).

106 Modern functional programming languages such as Haskell are in fact im-
107 plementations of typed lambda calculus with a flexible type system. Such
108 languages have polymorphic types, which means that functions and opera-

109 tions can apply generically to data. E.g., the operation that joins two lists has
 110 as its only requirement that the lists are of the same type a — where a can
 111 be the type of integers, the type of characters, the type of lists of characters,
 112 or any other type — and it yields a result that is again a list of type a .

113 This chapter will demonstrate, among other things, that implementing a
 114 Montague style fragment in a functional programming language with flexible
 115 types is a breeze: Montague’s underlying representation language is typed
 116 lambda calculus, be it without type flexibility, so Montague’s specifications
 117 of natural language fragments in PTQ Montague (1973) and UG Montague
 118 (1974b) are in fact already specifications of functional programs. Well, almost.

119 *Unification versus function composition in logical form construction*

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

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

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

139 *Literate Programming*

140 This Chapter is written in so-called literate programming style. Literate pro-
 141 gramming, as advocated by Donald Knuth in Knuth (1992), is a way of writing
 142 computer programs where the first and foremost aim of the presentation of a
 143 program is to make it easily accessible to humans. Program and documenta-
 144 tion are in a single file. In fact, the program source text is extracted from the
 145 L^AT_EX source text of the chapter. Pieces of program source text are displayed
 146 as in the following Haskell module declaration for this Chapter:

```
module IST where

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

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

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

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

162 *The role of models in computational semantics*

163 If one looks at computational semantics as an enterprise of constructing logical
164 forms for natural language sentences to express their meanings, then this may
165 seem a rather trivial exercise, or as Stephen Pulman once phrased it, an
166 “exercise in typesetting”. “John loves Mary” gets translated into $L(j, m)$,
167 and so what? The point is that $L(j, m)$ is a predication that can be checked
168 for truth in an appropriate formal model. Such acts of model checking are
169 what computational semantics is all about. If one implements computational
170 semantics, one implements appropriate models for semantic interpretation as
171 well, plus the procedures for model checking that make the computational
172 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 $\mathcal{P}D \rightarrow \mathcal{P}D \rightarrow \{0, 1\}$. In Montague style, elements of D have type e and the type of truth values is denoted t , so this becomes: $(e \rightarrow t) \rightarrow (e \rightarrow t) \rightarrow t$. 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 be specified as `[n..m]`. So here goes:

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

```

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

```

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

```

209     IST> intDET [1..] All (> 2) (> 3)
210     False
211     *IST> intDET [1..] All (> 3) (> 2)
212     ...

```

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

216 A direct interpretation instruction for “Most” is given by:

```

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

```

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

222 *Translation into logical form*

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

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

```

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

```

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

```

ix = Id "x" 0
233 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:

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

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

```

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

```

237 Some examples of variable terms:

```

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

```

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

```

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

```

241 Some examples of function symbols:

```

s   = Struct "s"
242 t   = Struct "t"
u   = Struct "u"

```

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

```

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

```

247 The function *isVar* checks whether a term is a variable; it uses the type
 248 *Bool* for Boolean (true or false). The type specification `Term -> Bool` says

249 that *isVar* is a classifier of terms. It classifies the the terms that start with
 250 **Var** as variables, and all other terms as non-variables.

```

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

```

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

```

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

```

257 This gives (you should check this for yourself):

```

258   *IST> isGround zero
259   True
260   *IST> isGround five
261   True
262   *IST> isGround (s[x])
263   False

```

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

```

269   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:

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

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

```

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

```

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

278 Before we can use the data type of formulas, we have to address a syntactic
 279 issue. The determiner expression is translated into a logical form construction
 280 recipe, and this recipe has to make sure that variables bound by a newly
 281 introduced generalized quantifier are bound properly. The definition of the
 282 `fresh` function that takes care of this can be found in the appendix. It is used
 283 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
284 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])

```

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

290 Of course, the task of providing formulas of the form $All v \phi_1\phi_2$ or the
 291 form $Most v \phi_1\phi_2$ with the correct interpretation is now shifted to the truth
 292 definition for the logical form language. We will turn to this in the next
 293 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 $a \rightarrow \text{Bool}$), relations (expressions of type $a \rightarrow a \rightarrow \text{Bool}$), higher order relations (expressions of type $(a \rightarrow \text{Bool}) \rightarrow (a \rightarrow \text{Bool}) \rightarrow \text{Bool}$), 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 $T \rightarrow F$ expresses an LF property, the type $T \rightarrow T \rightarrow F$ an LF relation, the type $(T \rightarrow F) \rightarrow (T \rightarrow F) \rightarrow F$ 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 $(T \rightarrow F) \rightarrow (T \rightarrow F) \rightarrow F$.

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..]

```

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

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

333 Let V be the set of variables of the language. A function $g: V \rightarrow D$ is
 334 called a *variable assignment* or *valuation*.

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

338 A variable assignment, in the implementation, is a function of type
 339 `Id -> a`, where `a` is the type of the domain of interpretation. The term lookup
 340 function takes a function symbol interpretation (type `FInterp a`) and vari-
 341 able assignment (type `Id -> a`) as inputs, and constructs a term assignment
 342 (type `Term -> a`), as follows.

```
343 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)
```

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

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

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

```
359 *IST> tVal fint1 (\ _ -> 0) five
360 5
```

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

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

367 Let g be a variable assignment or valuation. We use $g[v := d]$ for the
 368 valuation that is like g except for the fact that v gets value d (where g might
 369 have assigned a different value). For example, let $D = \{1, 2, 3\}$ be the domain
 370 of discourse, and let $V = \{v_1, v_2, v_3\}$. Let g be given by $g(v_1) = 1, g(v_2) =$
 371 $2, g(v_3) = 3$. Then $g[v_1 := 2]$ is the valuation that is like g except for the fact
 372 that v_1 gets the value 2, i.e. the valuation that assigns 2 to v_1 , 2 to v_2 , and 3
 373 to v_3 .

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

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

376 Let $M = (D, I)$ be a model for language L , i.e., D is the domain of
 377 discourse, I is an interpretation function for predicate letters and function
 378 symbols. Let g be a variable assignment for L in M . Let F be a formula of
 379 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 $\llbracket t \rrbracket_g^I$ as the term interpretation of t under I and g . With
 this notation, Tarski's truth definition can be stated as follows:

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

380 What we have presented just now is a recursive definition of truth for our
 381 logical form language. The 'relation specified by Q ' in the last clause refers to
 382 the generalized quantifier interpretations for *all*, *some*, *no* and *most*. Here is
 383 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
384 qRel No xs ys = not (qRel Some xs ys)
qRel Most xs ys =
    length (intersect xs ys) > length (xs \\ ys)

```

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

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

```

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

```

398 The evaluation function is defined for all types `a` that belong to the class `Eq`.
399 The assumption that the type `a` of the domain of evaluation is in `Eq` is needed
400 in the evaluation clause for equalities. The evaluation function takes a universe
401 (represented as a list, `[a]`) as its first argument, an interpretation function
402 for relation symbols (`Interp a`) as its second argument, an interpretation
403 function for function symbols as its third argument, a variable assignment
404 (`Id -> a`) as its fourth argument, and a formula as its fifth argument. The
405 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)
406 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

```

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

```

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

```

412 We need an interpretation for the predicates “Number” and “Even”. We
413 also throw in an interpretation for “Less than”:

```

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

```

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

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

```

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

```

424 Again we see a distinction between syntax (expressions like “plus” and
425 “times”) and semantics (Haskell operations like `+` and `*`).


```

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

```

428 This example uses a variable assignment $_ \rightarrow 0$ that maps any variable
429 to 0.

430 Now suppose we want to evaluate the following formula:

```

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

```

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

```

435 *IST> eval naturals int0 fint0 (\ _ -> 0) form1
436 ...

```

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

440 Suppose we want to define the relation “divides”. A natural number x
441 divides a natural number y if there is a number z with the property that
442 $x * z = y$. This is easily defined, as follows:

```

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

```

444 This gives:

```

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

```

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

451 *Interpretation of Natural Language Fragments*

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

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

```

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

```

460 Now we can define entities as follows:

```

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

```

462 Now, proper names will simply be interpreted as entities.

```

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

```

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

467 Let's define a type for predications:

```

468 type Pred a = [a] -> Bool

```

469 Some example properties:

```

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

```

471 Some example binary relations:

```

472 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)]

```

473 And here is an example of a ternary relation:

```

474 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)]

```

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

477 *Operations on predications*

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

```

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

```

482 Let's check this out:

```

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

```

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

```

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

```

493 *Reflexivization*

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

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

(1) *Alice loved herself.*

502 The reflexive *herself* is interpreted as a function that takes the two-place
 503 predicate *loved* as an argument and turns it into a one-place predicate, which

504 takes the subject as an argument, and expresses that this entity loves itself.
 505 This can be achieved by the following function `self`.

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

507 Here is an example application:

```
508 *IST> :t (self love)
509 (self love) :: Pred Entity
510 *IST> :t \ x -> self love [x]
511 \ x -> self love [x] :: Entity -> Bool
512 *IST> filter (\ x -> self love [x]) entities
513 [A]
```

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

Carol believed that Alice loved herself. (2)

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

**Herself loved Alice.* (3)

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

525 Reflexive pronouns can also be used to reduce ditransitive verbs to transi-
 526 tive verbs, in two possible ways: the reflexive can be the direct object or the
 527 indirect object:

Alice introduced herself to Bob. (4)

Bob gave the book to himself. (5)

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

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

531 *Quantifier scoping*

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

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

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

544

4 Example: Implementing Syllogistic Inference

545

546

547

548

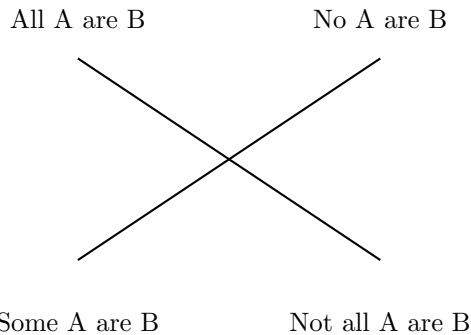
549

550

551

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:



552

553

554

555

556

557

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):

$Q ::=$ Are all PN PN?
 | Are no PN PN?
 | Are any PN PN?
 | Are any PN not PN?
 | What about PN?

558

$S ::=$ All PN are PN.
 | No PN are PN.
 | Some PN are PN.
 | Some PN are not PN.

559

560

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

- 561 • **ALL**: Set inclusion
- 562 • **SOME**: Non-empty set intersection
- 563 • **NOT ALL**: Non-inclusion
- 564 • **NO**: Empty intersection

565 Set inclusion: $A \subseteq B$ holds if and only if every element of A is an element
 566 of B . Non-empty set intersection: $A \cap B \neq \emptyset$ if and only if there is some
 567 $x \in A$ with $x \in B$. Non-empty set intersection can be expressed in terms of
 568 inclusion, negation and complementation, as follows: $A \cap B \neq \emptyset$ if and only if
 569 $A \not\subseteq \bar{B}$.

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

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

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

583 *Literals, Clauses, Clause Sets*

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

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

Represent the propositional formula

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

as the following clause set:

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

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

Unit Propagation

589 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

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

yields

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

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

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

590 The *Horn fragment* of propositional logic consists of all clause sets where
 591 every clause has *at most one positive literal*. Satisfiability for syllogistic forms
 592 containing exactly one existential statement translates to the Horn fragment
 593 of propositional logic. HORNSAT is the problem of testing Horn clause sets
 594 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.

596 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
  
```

598 We can represent a clause as a list of literals:

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

600 The names occurring in a list of clauses:

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

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

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

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

605 The property of being a unit clause:

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

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

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

610 The implementation uses an auxiliary function `prop` with three arguments.
611 The first argument gives the literals that are currently mapped to True, the

612 second argument gives the literals that occur in unit clauses, the third argu-
 613 ment 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,closures)
    prop xs (y:ys) clauses =
614     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

```

615 *Knowledge bases*

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

```

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

```

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

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

```

type Class = Lit

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

```

625 Statements and queries according to the grammar given above:

```

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

```

627 A statement display function is given in the appendix. Statement classifi-
628 cation:

```

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

```

630 Universal fact to statement. An implication $p \rightarrow q$ is represented as a
631 clause $\{\neg p, q\}$, and yields a universal statement “All p are q ”. An implication
632 $p \rightarrow \neg q$ is represented as a clause $\{\neg p, \neg q\}$, and yields a statement “No p are
633 q ”.

```

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

```

635 Existential fact to statement. A conjunction $p \wedge q$ is represented as a clause
636 set $\{\{p\}, \{q\}\}$, and yields an existential statement “Some p are q ”. A conjunc-
637 tion $p \wedge \neg q$ is represented as a clause set $\{\{p\}, \{\neg q\}\}$, and yields a statement
638 “Some p are not q ”.

```

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

```

640 Query negation:

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

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

666 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)
667 derive kb (AreAny as bs) = bs 'elem' (intersectionsets as kb)
derive kb (AnyNot as bs) = (neg bs) 'elem'
                           (intersectionsets as kb)

```

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

```

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

```

674 Update with an ‘All’ statement. The update function checks for possible
675 inconsistencies. E.g., a request to add an $A \subseteq B$ fact to the knowledge base
676 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)
677 | otherwise = Just (([as',bs]:xs,yss),True)
  where
    as' = neg as
    bs' = neg bs

```

678 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)
679 | 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)
680 | 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)
681 | otherwise = Just ((xs,[[as],[bs']] :yss),True)
where
  bs' = neg bs

```

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

688 There is also a link to the “natural logic for natural language” enterprise:
689 the logical forms for syllogistic reasoning are very close to the surface forms
690 of the sentences. The Chapter on Natural Logic in this Handbook gives more
691 information. All in all, reasoning engines like this one are relevant for rational
692 reconstructions of cognitive processing. The appendix gives the code for con-
693 structing a knowledge base from a list of statements, and updating it. Here
694 is a chat function that starts an interaction from a given knowledge base and
695 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:"
696   str <- getLine
      if str == "" then return ()
      else do
        handleCases kb str
        chat
```

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

699

5 Implementing Fragments of Natural Language

700

701

702

703

704

705

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:

706

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

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

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

707

708

709

710

711

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:

712

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

713

714

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:

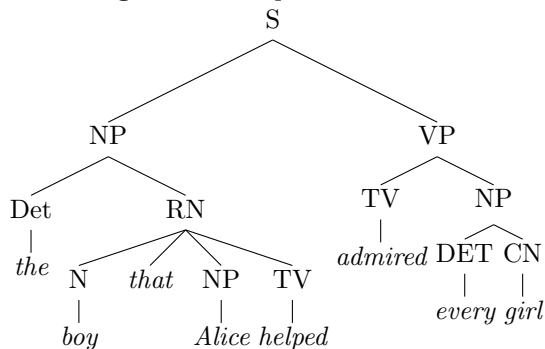

```

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
715 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

```

716 Look at this as a definition of syntactic structure trees. The structure for
717 *The boy that Alice helped admired every girl* is given in Figure 1, with the
718 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)))

```

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

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

727 In any case, VP representations will have a functional type, for VPs de-
 728 note properties. A reasonable type for the function that represents a VP is
 729 $Term \rightarrow Formula$. If we feed it with a term, it will yield a logical form. Proper
 730 names now can get the type of terms. Take the example *Alice laughed*. The
 731 verb *laughed* gets represented as the function that maps the term x to the
 732 formula $Atom\ "laugh"\ [x]$. Therefore, we get an appropriate logical form for
 733 the sentence if x is a term for *Alice*.

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

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

740 Next, NP-representations are of type $(Term \rightarrow Formula) \rightarrow Formula$.

```
741 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)
```

742 Verb phrase representations are of type $Term \rightarrow Formula$.

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

744 Representing a function that takes two arguments can be done either by
 745 means of $a \rightarrow a \rightarrow b$ or by means of $(a,a) \rightarrow b$. A function of the first
 746 type is called *curried*, a function of the second type *uncurried*.

747 We assume that representations of transitive verbs are uncurried, so they
 748 have type $(Term,Term) \rightarrow Formula$, where the first term slot is for the sub-
 749 ject, and the second term slot for the object. Accordingly, the representations
 750 of ditransitive verbs have type

(Term,Term,Term) -> Formula

751

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:

752

753

754

```

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)))

```

755

Representations for transitive verbs are:

756

```

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]

```

757

Ditransitive verbs:

758

```

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]

```

759

Common nouns have the same type as VPs.

760

```

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

```

761

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

762

763

```

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))]

```

764

We end with some examples:

765

```

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 interpretation 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)
```

849 The restriction of a partition to a domain:

```

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

```

851 An infinite number of agents, with names *a, b, c, d, e* for the first five of
852 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

853 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

```

854 A datatype for epistemic models:

```

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

```

856 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]

```

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

861 Extracting an epistemic relation from a model:

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

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

863 This gives:

```
864 *IST> rel a example
865 [[0],[1],[2],[3]]
866 *IST> rel c example
867 [[0,1,2,3]]
868 *IST> rel d example
869 *** Exception: item not found
```

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

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

873 A useful abbreviation:

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

875 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)
876 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)

```

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

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

```

upd_pa :: Ord state =>
    EpistM state -> Form state -> EpistM state
upd_pa m@(Mo states agents rels actual) f =
882 (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' ]

```

883 A series of public announcement updates:

```

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

```

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

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

892 P only of $p = xy$. These announcements remain private. You are
 893 required to determine the pair (x, y) . He acts as said. The following
 894 conversation now takes place:

- 895 (1) P says: “I do not know the pair.”
 896 (2) S says: “I knew you didn’t.”
 897 (3) P says: “I now know it.”
 898 (4) S says: “I now also know it.”
 899 Determine the pair (x, y) .

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

904 The list of candidate pairs:

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

906 The initial epistemic model is such that a (representing S) cannot dis-
 907 tinguish number pairs with the same sum, and b (representing P) cannot
 908 distinguish number pairs with the same product. Instead of using a valuation,
 909 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 ] ) ]
```

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

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

913 To check this statement is expensive. A computationally cheaper equiva-
 914 lent statement is the following (see Ditmarsch *et al.* (2005)).

```

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

```

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

```

921 k_a_statement_1e = Kn a statement_1e

```

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

```

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

```

925 Equivalently, but computationally more efficient:

```

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

```

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

```

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

```

930 In the computationally optimized version:

```

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

```

932 The solution:

```

933 solution = upds_pamsp
           [k_a_statement_1e,statement_2e,statement_3e]

```

934 This is checked in a matter of minutes:

```

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

```

937 **8 Resources**938 *Code for this Chapter*

939 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
940 Haskell system, which can be downloaded from www.haskell.org. This site
941 also gives many interesting Haskell resources.
942

943 *Epistemic model checking*

944 More information on epistemic model checking can be found in the documen-
945 tation of the epistemic model checker DEMO. See Eijck (2007).

946 *Link for Computational Semantics With Functional Programming*

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

949 *Further computational semantics links*

950 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.
951
952
953

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:

```

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 ++ "."
964 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:

```

makeKB :: [Statement] -> Maybe KB
makeKB = makeKB' ([], [])
  where
966   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:

```

preprocess :: String -> [String]
968 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 stupid\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 stupid\n"
      ++ "some men are not stupid"

```

975 Reading a knowledge base from disk:

```

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

```

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

```

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

```

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

```

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) ]
980     ++
    [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"

```

983

References

- 984
- 985 Alshawi, H. (ed.) (1992), *The Core Language Engine*, MIT Press, Cambridge Mass,
 986 Cambridge, Mass., and London, England.
- 987 Alshawi, H. & J. van Eijck (1989), Logical forms in the core language engine, in
 988 *Proceedings of the 27th Congress of the ACL*, ACL, Vancouver.
- 989 Aumann, R.J. (1976), Agreeing to disagree, *Annals of Statistics* 4(6):1236–1239.
- 990 Barwise, J. & R. Cooper (1981), Generalized quantifiers and natural language, *Lin-*
 991 *guistics and Philosophy* 4:159–219.
- 992 Battigalli, P. & G. Bonanno (1999), Recent results on belief, knowledge and the
 993 epistemic foundations of game theory, *Research in Economics* 53:149–225.
- 994 van Benthem, J. (2011), *Logical Dynamics of Information and Interaction*, Cam-
 995 bridge University Press.
- 996 Blackburn, P. & J. Bos (2005), *Representation and Inference for Natural Language;*
 997 *A First Course in Computational Semantics*, CSLI Lecture Notes.
- 998 Biring, D. (2005), *Binding Theory*, Cambridge Textbooks in Linguistics, Cambridge
 999 University Press.
- 1000 Ditmarsch, Hans van, Ji Ruan, & Rineke Verbrugge (2005), Model checking sum
 1001 and product, in Shichao Zhang & Ray Jarvis (eds.), *AI 2005: Advances in Arti-*
 1002 *ficial Intelligence: 18th Australian Joint Conference on Artificial Intelligence*,
 1003 Springer-Verlag GmbH, volume 3809 of *Lecture Notes in Computer Science*,
 1004 (790–795).
- 1005 Ditmarsch, H.P. van, W. van der Hoek, & B. Kooi (2006), *Dynamic Epistemic Logic*,
 1006 volume 337 of *Synthese Library*, Springer.
- 1007 Eijck, Jan van (2007), DEMO — a demo of epistemic modelling, in Johan van Ben-
 1008 them, Dov Gabbay, & Benedikt Löwe (eds.), *Interactive Logic — Proceedings of*
 1009 *the 7th Augustus de Morgan Workshop*, Amsterdam University Press, number 1
 1010 in *Texts in Logic and Games*, (305–363).
- 1011 Eijck, Jan van & Christina Unger (2010), *Computational Semantics with Functional*
 1012 *Programming*, Cambridge University Press.
- 1013 Fagin, R., J.Y. Halpern, Y. Moses, & M.Y. Vardi (1995), *Reasoning about Knowl-*
 1014 *edge*, MIT Press.
- 1015 Gärdenfors, P. (1988), *Knowledge in Flux: Modelling the Dynamics of Epistemic*
 1016 *States*, MIT Press, Cambridge Mass.
- 1017 Hintikka, J. (1962), *Knowledge and Belief: An Introduction to the Logic of the Two*
 1018 *Notions*, Cornell University Press, Ithaca N.Y.
- 1019 Hughes, J. (1989), Why functional programming matters, *The Computer Journal*
 1020 32(2):98–107, ISSN 0010-4620, doi:10.1093/comjnl/32.2.98.
- 1021 Knuth, D.E. (1992), *Literate Programming*, CSLI Lecture Notes, no. 27, CSLI, Stan-
 1022 ford.
- 1023 Montague, R. (1973), The proper treatment of quantification in ordinary English,
 1024 in J. Hintikka (ed.), *Approaches to Natural Language*, Reidel, (221–242).
- 1025 Montague, R. (1974a), English as a formal language, in R.H. Thomason (ed.), *Formal*
 1026 *Philosophy; Selected Papers of Richard Montague*, Yale University Press, New
 1027 Haven and London, (188–221).
- 1028 Montague, R. (1974b), Universal grammar, in R.H. Thomason (ed.), *Formal Philos-*
 1029 *ophy; Selected Papers of Richard Montague*, Yale University Press, New Haven
 1030 and London, (222–246).

1031 Tarski, A. (1956), The concept of truth in the languages of the deductive sciences, in
1032 J. Woodger (ed.), *Logic, Semantics, Metamathematics*, Oxford, first published
1033 in Polish in 1933.

Index

1036	Aristotle, 22	1064	Horn clauses, 3
1037	Backus Naur Form, 3	1065	HORNSAT algorithm, 24
1038	BNF, 3	1066	imperative programming, 2
1039	Boolean, 7	1067	individual concept, 37
1040	Cam1, 4	1068	intension, 37
1041	Church, Alonzo, 3	1069	Kowalski, Robert, 3
1042	clause, 23	1070	lazy evaluation, 4
1043	clause set, 23	1071	LF function
1044	co-recursion, 2	1072	1fN, 35
1045	Colmerauer, Alain, 3	1073	1fNP, 34
1046	curried function, 34	1074	1fRN, 35
1047	Curry, Haskell B., 4	1075	1fS, 34
1048	datalog, 3	1076	1fTV, 35
1049	declarative programming, 2	1077	1fVP, 34–35
1050	DEMO, 44	1078	Lisp, 4
1051	Ditmarsch, Hans van, 42	1079	literal, 23
1052	domain of discourse, 12	1080	McCarthy, John, 41
1053	empty list, 7	1081	ML, 4
1054	epistemic logic, 38	1082	Moss, Larry, 22
1055	epistemic model, 39	1083	natural logic, 22
1056	equivalence relation, 38	1084	non-strict evaluation, 4
1057	evaluation function, 15	1085	Ocaml, 4
1058	existential import, 22	1086	partition, 38
1059	extension, 37	1087	Prolog, 3
1060	first order model, 12	1088	proposition, 37
1061	fixpoint computation, 3	1089	Pulman, Stephen, 6
1062	Freudenthal, Hans, 41	1090	Pure lambda calculus, 3
1063	Haskell, 4		

1091	rigid designator, 37	1100	Tarski, Alfred, 12
1092	Ruan, Ji, 42	1101	truth definition of Tarski, 12
1093	Russell's paradox, 3	1102	Turing, Alan, 4
1094	Scheme, 4	1103	uncurried function, 34
1095	self, 20	1104	unification, 3
1096	SLD resolution, 3	1105	unit propagation, 23
1097	square of opposition, 22	1106	valuation, 13
1098	Sum and Product, 41	1107	variable assignment, 13
1099	syllogistics, 22	1108	Verbrugge, Rineke, 42

