Form and Content

Jan van Eijck

CWI, Amsterdam and Uil-OTS, Utrecht

jve@cwi.nl

LOT Summer School, June 17, 2009

Abstract

We look at the distinction between form and content, or syntax and semantics, or structure and meaning. After making this distinction a bit more precise, we study the composition of meaning.

Module Declaration

```
module FormContent

where
import List
import Char
```

Form

Form is given by syntax. As an example, we give a datatype for syntax trees in Haskell.

```
data Sent = Sent NP VP
     deriving (Eq,Show)

data NP = Ann | Mary | Bill | Johnny
        | NP1 DET CN | NP2 DET RCN
     deriving (Eq,Show)

data DET = Every | Some | No | The | Most
         | Atleast Int
     deriving (Eq,Show)

data CN = Man | Woman | Boy | Person
        | Thing | House
     deriving (Eq,Show)
```

```
data RCN = CN1 CN VP | CN2 CN NP TV
     deriving (Eq,Show)

data VP = Laughed | Smiled | VP1 TV NP
     deriving (Eq,Show)

data TV = Loved | Respected | Hated | Owned
     deriving (Eq,Show)
```

Content

It is hard to say what content is. But the relevant notion is sameness of content.

Replace the question 'What is the meaning of a sentence?' by the more precise question 'When do two sentences express the same meaning'?

Let us restrict attention to declarative sentences. Declarative sentences are sentences that can be either true or false in a given context.

'It is raining today in Utrecht' and 'I am Dutch' are declarative sentences. If they are uttered, the context of utterance fixes the meaning of 'today' and 'I', and the uttered sentences are either true or false in that context.

'Let's try to be smarter next time' is not a declarative sentence. 'Is drinking coffee bad for you?' is not a declarative sentence either.

Sameness of Meaning

'Jan van Eijck is Dutch' and 'I am Dutch' do not have the same meaning, for if I utter them they are both true, and if someone from abroad utters them one will be true and the other false.

'Jan van Eijck is Dutch' and 'Jan van Eijck is Nederlander' have the same meaning, as have 'Cinderella est belle' and 'Assepoester is mooi'.

To check for 'Sameness of meaning' one has to interpret sentences in many different situations, and check if the resulting truth values are always the same.

But what does 'interpretation of a sentence in a situation' mean?

To replace the intuitive understanding by a precise understanding we can look at formal examples: the language of predicate logic and its semantics, or the Haskell language, and its interpretation.

Basic Sentences in Predicate Logic, and in Haskell

Predicate logic is the logic of predicates. A predicate is a word that combines with a certain number of proper names to form a basic sentence.

Example:

$P(a, b)$

This expresses that a, b are in the relation given by P. But what is "the relation given by P"? That depends on the interpretation.

Interpretation

What is an interpretation?

An interpretation for predicates consists of a domain of discourse D and an instruction for connecting the predicates to the domain. If P is a predicate that takes a pair of names to form a basic sentence, then an interpretation for P is a binary relation on D.

The number of names that a basic predicate needs to form a basic sentence is called the arity of the predicate. Predicates that take one name are called unary. Their interpretation is a subset of the domain of discourse D. Predicates that take two names are called binary, predicates that take three names ternary.

Predicate Logic in Haskell

The domain of discourse is some Haskell type. Let us say the type of Integers.

Predicates are properties of integers, such as `odd`, `even`, `threefold`, `(>0)`, and relations such as `(>)`, `(<=)`.

Logical operations on predicates are negation, conjunction, disjunction.

'even or threefold' becomes `\ x -> even x || rem x 3 == 0`.

'not even' becomes `not . even` or `\ x -> not (even x)`.

Quantifications are 'some integers in [1..100] are even', or 'all integers in [1..] are positive'.

Examples of Quantifications in Haskell

'some integers in [1..100] are even'

```
FormContent> any even [1..100]
True
```

'all integers in [1..100] are positive':

```
FormContent> all (>0) [1..100]
True
FormContent> all (>0) [1..]
{Interrupted!}
```

Question: what is the type of `all` and `any` ?

Question: does a quantification over an infinite list (like `[1..]`) always run forever?

A Domain of Discourse in Haskell

```
data Entity = A | B | C | D | E | F | G
            | H | I | J | K | L | M | N
            | O | P | Q | R | S | T | U
            | V | W | X | Y | Z | Unspec
     deriving (Eq,Bounded,Enum)
```

Because `Entity` is a bounded and enumerable type, we can put all of its elements in a finite list:

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

A Show Function for Entities

```
instance Show Entity where
  show (A) = "A"; show (B) = "B"; show (C) = "C";
  show (D) = "D"; show (E) = "E"; show (F) = "F";
  show (G) = "G"; show (H) = "H"; show (I) = "I";
  show (J) = "J"; show (K) = "K"; show (L) = "L";
  show (M) = "M"; show (N) = "N"; show (O) = "O";
  show (P) = "P"; show (Q) = "Q"; show (R) = "R";
  show (S) = "S"; show (T) = "T"; show (U) = "U";
  show (V) = "V"; show (W) = "W"; show (X) = "X";
  show (Y) = "Y"; show (Z) = "Z"; show (Unspec)= "*"
```

```
FormContent> entities
[A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z,*]
```

Relations on the Domain

Example relation:

```
rel1 :: Entity -> Entity -> Bool
rel1 A A = True
rel1 B A = True
rel1 D A = True
rel1 C B = True
rel1 C C = True
rel1 C D = True
rel1 _ _ = False
```

```
FormContent> filter (\x -> rel1 x A) entities
[A,B,D]
```

Arity Reduction on Binary Relations

```
self :: (a -> a -> b) -> a -> b
self = \ f x -> f x x
```

The following definition picks the reflexive part out of `rel1`:

```
rel2 = self rel1
```

```
FormContent> filter (self rel1) entities
[A,C]
```

Representing a Model

Interpretations for proper names:

```
ann, bill, lucy, mary, johnny :: Entity
ann    = A; bill   = B; lucy = L
mary   = M; johnny = J
```

Conversion function

For easy specification of (unary) predicates:

```
list2pred :: Eq a => [a] -> a -> Bool
list2pred = flip elem
```

This uses:

```
flip           :: (a -> b -> c) -> b -> a -> c
flip f x y      = f y x
```

Interpretations for Predicates

```
man, boy, woman, tree, house :: Entity -> Bool
leaf, stone, gun, person, thing :: Entity -> Bool
man    = list2pred [B,J]
woman  = list2pred [A,C,M,L]
boy    = list2pred [J]
tree   = list2pred [T,U,V]
house  = list2pred [H,K]
leaf   = list2pred [X,Y,Z]
stone  = list2pred [S]
gun    = list2pred [G]
```

A person is a man or a woman, and a thing is everything which is neither a person nor the special object `Unspec`:

```
person = \ x -> (man x || woman x)
thing  = \ x -> not (person x || x == Unspec)
```

Meanings for Intransitive Verbs

Same type as CN meanings:

```
laugh, smile :: Entity -> Bool
laugh = list2pred [M]
smile = list2pred [A,B,J,M]
```

Binary Relations: Meanings for Transitive Verbs

```
love, respect, hate, own, wash, shave, drop0
          :: (Entity, Entity) -> Bool
love    = list2pred
          [(B,M),(J,M),(J,J),(M,J),(A,J),(B,J)]
respect = list2pred [(x,x)
                        | x <- entities, person x ]
hate    = list2pred [(x,B)
                         | x <- entities, woman x ]
own     = list2pred [(M,H)]
wash    = list2pred [(A,A),(A,J),(L,L),(B,B),(M,M)]
shave   = list2pred [(A,J),(B,B)]
drop0   = list2pred [(T,X),(U,Y),(U,Z),(Unspec,V)]
```

Ternary Relations

```
break0, kill ::
          (Entity, Entity, Entity) -> Bool
break0 = list2pred [(M,V,S), (J,W,G)]
kill   = list2pred
          [(M,L,G), (Unspec,A,D), (Unspec,J,Unspec)]
```

The verbs give and sell are also interpreted as ternary relations.

```
give, sell :: (Entity, Entity, Entity) -> Bool
give = list2pred [(M,V,L), (L,G,M)]
sell = list2pred [(J,J,M), (J,T,M), (A,U,M)]
```

Conversions for Ternary Relations

```
curry3 :: ((a,b,c) -> d)
          -> a -> b -> c -> d
curry3 f x y z = f (x,y,z)

uncurry3     ::
  (a -> b -> c -> d) -> ((a,b,c) -> d)
uncurry3 f (x,y,z)  = f x y z
```

Semantic Interpretation: Compositionality

Fix a situation: a domain of discourse with properties and relations defined on it. Logicians call this a model.

Next, fix the interpretation of individual words, by linking proper names to entities in the domain of discourse, intransitive verbs and common nouns to properties, transitive verbs to binary relations, and so on.

Finally, define a composition function that computes the meanings of composite expressions from the meanings of their parts.

This is called: compositional interpretation.

Note: The semantic type of the interpretation depends on the syntactic category of the expression that gets interpreted.

Semantic Interpretation: Sentences

Syntactic categories get interpretations of appropriate types.

Type for the interpretation of sentences: `Bool`.

```
intSent :: Sent -> Bool
intSent (Sent np vp) = (intNP np) (intVP vp)
```

Semantic Interpretation: Noun Phrases

Type for the interpretation of NPs: `(Entity -> Bool) -> Bool`.

```
intNP :: NP -> (Entity -> Bool) -> Bool
intNP Ann = \ p -> p ann
intNP Mary = \ p -> p mary
intNP Bill = \ p -> p bill
intNP Johnny = \ p -> p johnny
intNP (NP1 det cn) = (intDET det) (intCN cn)
intNP (NP2 det rcn) = (intDET det) (intRCN rcn)
```

Semantic Interpretation: Verb Phrases

```
intVP :: VP -> Entity -> Bool
intVP Laughed = laugh
intVP Smiled = smile
```

```
intVP (VP1 tv np) =
   \ subj ->
       intNP np (\ obj -> intTV tv (subj,obj))
```

Semantic Interpretation: Transitive Verbs

```
intTV :: TV -> (Entity,Entity) -> Bool
intTV Loved     = love
intTV Respected = respect
intTV Hated     = hate
intTV Owned     = own
```

Semantics Interpretation: Common Nouns

Similar to that of Verb Phrases:

```
intCN :: CN -> Entity -> Bool
intCN Man = man
intCN Boy = boy
intCN Woman = woman
intCN Person = person
intCN Thing = thing
intCN House = house
```

Semantic Interpretation of Determiners: Some and Every

Type of interpretation function:

```
intDET :: DET -> (Entity -> Bool)
          -> (Entity -> Bool) -> Bool
```

```
intDET Some p q = any q (filter p entities)

intDET Every p q = all q (filter p entities)
```

Semantic Interpretation of Determiners: The

The interpretation of The consists of two parts:

1. a check that the CN property is unique, i.e., that it is true of precisely one entity in the domain,
2. a check that the CN and the VP property have an element in common, in other words, the Some check on the two properties.

```
intDET The p q = singleton plist && q (head plist)
          where
              plist = filter p entities
              singleton [x] = True
              singleton _   = False
```

Semantic Interpretation of Determiners: No

The interpretation of No is just the negation of the interpretation of Some:

```
intDET No p q = not (intDET Some p q)
```

Semantic Interpretation of Determiners: Most

The interpretation of Most compares the length of the list of entities satisfying the first argument (the restrictor argument) with the length of the list of entities satisfying the second argument (the body argument).

```
intDET Most p q = length pqlist >
                  length (plist \\ qlist)
  where
  plist  = filter p entities
  qlist  = filter q entities
  pqlist = filter q plist
```

Exercise: Implement the interpretation function for `(Atleast n)`.

Semantic Interpretation of Relativized CNs

Relativised common nouns of the form that CN VP:

```
intRCN :: RCN -> Entity -> Bool
intRCN (CN1 cn vp) =
        \ e -> ((intCN cn e) && (intVP vp e))
```

Relativised common nouns of the form that CN NP TV:

```
intRCN (CN2 cn np tv) = \ e ->
    ((intCN cn e) &&
     (intNP np (\ subj -> (intTV tv (subj,e)))))
```

Example Queries

```
FormContent> intSent (Sent (NP1 The Boy) Smiled)
True
FormContent> intSent (Sent (NP1 The Boy) Laughed)
False
FormContent> intSent (Sent (NP1 Some Man) Laughed)
False
FormContent> intSent (Sent (NP1 No Man) Laughed)
True
FormContent> intSent (Sent (NP1 Some Man)
                     (VP1 Loved (NP1 Some Woman)))
True
FormContent> intSent (Sent (NP2 No (CN1 Man (VP1 Loved Mary)
             Laughed)
True
```

After the break

More about logic . . .