Computational Semantics, Type Theory, and Functional Programming

I — Converting Montague Grammarians into Programmers

Jan van Eijck

CWI and ILLC, Amsterdam, Uil-OTS, Utrecht

LOLA7 Tutorial, Pecs

August 2002

Summary

- Functional Expressions and Types
- Type Theory, Type Polymorphism [Hin97]
- Functional Programming
- Representing Semantic Knowledge in Type Theory
- Building a Montague Fragment [Mon73]
- Datastructures for Syntax
- Semantic Interpretation

Functional Expressions and Types

According to Frege, the meaning of *John arrived* can be represented by a function argument expression Aj where A denotes a function and j an argument to that function.

The expression A does not reveal that it is supposed to combine with an individual term to form a formula (an expression denoting a truth value). One way to make this explicit is by means of lambda notation. The function expression of this example is then written as $(\lambda x.Ax)$.

It is also possible to be even more explicit, and write

$$\lambda x.(Ax) :: e \rightarrow t$$

to indicate the type of the expression, or even:

$$(\lambda x_e.A_{e\rightarrow t}x)_{e\rightarrow t}.$$

Definition of Types

The set of *types* over e, t is given by the following BNF rule:

$$T \ ::= \ e \mid t \mid (T \rightarrow T).$$

The basic type e is the type of expressions denoting individual objects (or *entities*). The basic type t is the type of formulas (of expressions which denote *truth values*). Complex types are the types of functions.

For example, $(e \rightarrow t)$ or $e \rightarrow t$ is the type of functions from entities to truth values.

In general: $T_1 \rightarrow T_2$ is the type of expressions denoting functions from denotations of T_1 expressions to denotations of T_2 expressions.

Picturing a Type Hierarchy

The types e and t are given. Individual objects or entities are objects taken from some domain of discussion D, so e type expressions denote objects in D. The truth values are $\{0,1\}$, so type t expression denotes values in $\{0,1\}$.

For complex types we use recursion. This gives:

$$D_e = D, D_t = \{0,1\}, D_{A\to B} = {D_B}^{D_A}.$$

Here ${D_B}^{D_A}$ denotes the set of all functions from D_A to D_B.

A function with range $\{0, 1\}$ is called a *characteristic function*, because it characterizes a set (namely, the set of those things which get mapped to 1).

If T is some arbitrary type, then any member of $D_{T\to t}$ is a characteristic function. The members of $D_{e\to t}$, for instance, characterize subsets of the domain of individuals D_e.

As another example, consider $D_{(e\to t)\to t}$. According to the type definition this is the domain of functions ${D_t}^{D_{e\to t}}$, i.e., the functions in $\{0, 1\}^{D_{e\to t}}$. These functions characterize sets of subsets of the domain of individuals D_e.

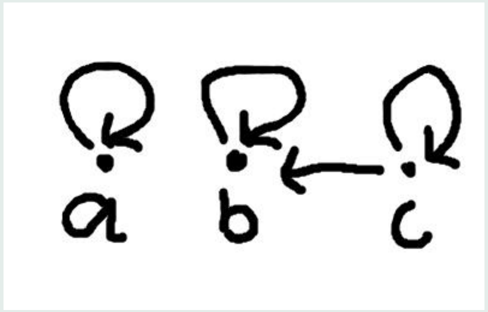
Types of Relations

As a next example, consider the domain $D_{e\to(e\to t)}$. Assume for simplicity that D_e is the set $\{a,b,c\}$. Then we have:

$$D_{e\to(e\to t)} = {D_{e\to t}}^{D_e} = (D_t^{D_e})^{D^e} = (\{0,1\}^{\{a,b,c\}})^{\{a,b,c\}}.$$

Example element of $D_{e\to(e\to t)}$:

$$\begin{array}{ccc}
a & \mapsto & \begin{pmatrix} a \mapsto 1 \\ b \mapsto 0 \\ c \mapsto 0 \end{pmatrix} \\
b & \mapsto & \begin{pmatrix} a \mapsto 0 \\ b \mapsto 1 \\ c \mapsto 1 \end{pmatrix} \\
c & \mapsto & \begin{pmatrix} a \mapsto 0 \\ b \mapsto 0 \\ c \mapsto 1 \end{pmatrix}
\end{array}$$



The elements of $D_{e\longrightarrow(e\longrightarrow t)}$ can in fact be regarded as functional encodings of two-placed relations R on D_e, for a function in $D_{e\longrightarrow(e\longrightarrow t)}$ maps every element d of D_e to (the characteristic function of) the set of those elements of D_e that are R-related to d.

Types for Propositional Functions

Note that $D_{t \to t}$ has precisely four members, namely:

identity	negation	constant 1	constant 0
$1 \mapsto 1$ $0 \mapsto 0$	$1 \mapsto 0$ $0 \mapsto 1$	$1 \mapsto 1$ $0 \mapsto 1$	$0 \mapsto 0$ $0 \mapsto 0$

The elements of $D_{t \to (t \to t)}$ are functions from the set of truth values to the functions in $D_{t \to t}$, i.e., to the set of four functions pictured above.

As an example, here is the function which maps 1 to the constant 1 function, and 0 to the identity:

$$\begin{array}{ccc} 1 & \mapsto & \begin{pmatrix} 1 \mapsto 1 \\ 0 \mapsto 1 \end{pmatrix} \\ 0 & \mapsto & \begin{pmatrix} 1 \mapsto 1 \\ 0 \mapsto 0 \end{pmatrix} \end{array}$$

We can view this as a ‘two step’ version of the semantic operation of taking a *disjunction*.

If the truth value of its first argument is 1, then the disjunction becomes true, and the truth value of the second argument does not matter (hence the constant 1 function).

$$1 \;\mapsto\; \begin{pmatrix} 1 \mapsto 1 \\ 0 \mapsto 1 \end{pmatrix}$$

If the truth value of the first argument is 0, then the truth value of the disjunction as a whole is determined by the truth value of the second argument (hence the identity function).

$$0 \;\mapsto\; \begin{pmatrix} 1 \mapsto 1 \\ 0 \mapsto 0 \end{pmatrix}$$

Types in Haskell

In this section we take a brief look at how types appear in the functional programming language Haskell [HFP96, JH$^+$99, Bir98, DvE02].

For Haskell, see: `www.haskell.org`

The text inside the boxes is literal Haskell code. For convenience we collect the code of this tuturial in a module. The present module is called *LOLA1*.

```
module LOLA1 where

import Domain
import Model
```

In Haskell, the type for truthvalues is called `Bool`. Thus, the type for the constant function that maps all truth values to 1 is `Bool -> Bool`. The truth value 1 appears in Haskell as `True`, the truth value 0 as `False`. The definitions of the constant `True` and the constant `False` function in Haskell run as follows:

```haskell
c1 :: Bool -> Bool
c1 _ = True

c0 :: Bool -> Bool
c0 _ = False
```

The definitions consist of a type declaration and a value specification. Whatever argument you feed `c1`, the result will be `True`. Whatever argument you give `c0`, the result will be `False`. The underscore _ is used for any value whatsoever.

An equivalent specification of these functions runs as follows:

```
c1 :: Bool -> Bool
c1 = \ p -> True

c0 :: Bool -> Bool
c0 = \ p -> False
```

The function in `Bool -> Bool` that swaps the two truth values is predefined in Haskell as `not`.

The identity function is predefined in Haskell as `id`. This function has the peculiarity that it has *polymorphic type*: it can be used as the identity for *any* type for which individuation makes sense.

In Haskell, the disjunction function will have type

```
Bool -> Bool -> Bool
```

This type is read as `Bool -> (Bool -> Bool)`.

Implementation of `disj`, in terms of `c1` and `id`:

```
disj :: Bool -> Bool -> Bool
disj True  = c1
disj False = id
```

Alternative:

```
disj :: Bool -> Bool -> Bool
disj = \ p -> if p then c1 else id
```

Type Polymorphism

The `id` function:

```
id :: a -> a
id x = x
```

A function for *arity reduction*:

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

This gives:

```
LOLA1> self (<=) 3
True
```

Properties, Relations

The polymorphic type of a property is `a -> Bool`.

Examples: `(>= 0)`, `(<> x)`, `even`.

The polymorphic type of a relation is `a -> b -> Bool`.

The polymorphic type of a relation over a single type is `a -> a -> Bool`.

Examples: `(<)`, `(<=)`, `(==)`.

List Types

Examples of list types:

- `[Int]` is the type of lists of integers,
- `[Char]` is the type of lists of characters, or strings,
- `[Bool]` is the type of lists of Booleans,
- and so on.

If a is a type, `[a]` is the type of lists over a.

Note that `[a]` is a polymorphic type.

`[]` is the empty list (of *any* type).

`(x:xs)` is the prototypical non-empty list.

The `head` of `(x:xs)` is `x`, the `tail` is `xs`.

List Comprehension

List comprehension is the list counterpart of set comprehension:

$$\{x \mid x \in A,\ P(x)\}$$

```
LOLA1> [ n |n <- [0..20], n `mod` 7 == 1 ]
[1,8,15]

LOLA1> [ 2^n | n <- [0..10] ]
[1,2,4,8,16,32,64,128,256,512,1024]

LOLA1> [ w ++ w | w <- ["ab","cd","eef"] ]
["abab","cdcd","eefeef"]
```

The `map` Function

The function `map` takes a function and a list and returns a list containing the results of applying the function to the individual list members.

If `f` is a function of type `a -> b` and `xs` is a list of type `[a]`, then `map f xs` will return a list of type `[b]`.

```
map :: (a -> b) -> [a] -> [b]
```

```
map f [] = []
map f (x:xs) = (f x) : map f xs
```

E.g., `map (^2) [1..9]` will produce the list of squares:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

The `filter` Function

The `filter` function takes a property and a list, and returns the sublist of all list elements satisfying the property.

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) | p x         = x : filter p xs
                | otherwise = filter p xs
```

```
LOLA1> filter even [1..10]
[2,4,6,8,10]
```

Function Composition

For the composition `f . g` to make sense, the result type of `g` should equal the argument type of `f`, i.e., if `f :: a -> b` then `g :: c -> a`.

Under this typing, the type of `f . g` is `c -> b`.

Thus, the operator `(.)` that composes two functions has the following type and definition:

```
(.) :: (a -> b) -> (c -> a) -> c -> b
(f . g) x = f (g x)
```

A Domain of Entities

To illustrate the type e, we construct a small example domain of entities consisting of individuals A, …, M, by declaring a datatype `Entity`.

```
module Domain where

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

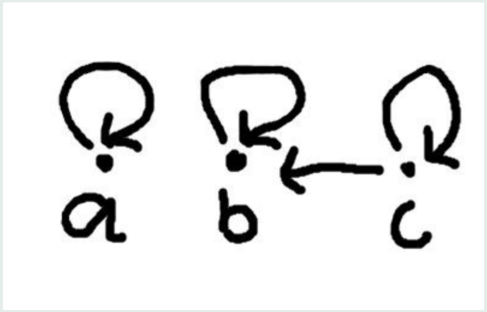
The stuff about `deriving (Eq,Bounded,Enum,Show)` is there to enable us to do equality tests on entities (`Eq`), to refer to A as the minimum element and M as the maximum element (`Bounded`), to enumerate the elements (`Enum`), and to display the elements on the screen (`Show`).

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

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

This gives:

```
CompSem1> entities
[A,B,C,D,E,F,G,H,I,J,K,L,M]
```



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

This gives:

```
LOLA1> r A B
False
LOLA1> r A A
True
```

Using lambda abstraction, we can select sublists of a list that satisfy some property we are interested in. Suppose we are interested in the property of being related by `r` to the element `B`. This property can be expressed using λ abstraction as $\lambda x.(Rb)x$. In Haskell, this same property is expressed as `(\ x -> r B x)`.

Using a property to create a sublist is done by means of the `filter` operation. Here is how we use `filter` to find the list of entities satisfying the property:

```
LOLA1> filter (\ x -> r B x) entities
[B,C]
```

With negation, we can express the complement of this property. This gives:

```
LOLA1> filter (\ x -> not (r B x)) entities
[A,D,E,F,G,H,I,J,K,L,M]
```

Here is how `disj` is used to express the property of being either related to a or to b:

```
LOLA1> filter (\ x -> disj (r A x) (r B x)) entities
[A,B,C]
```

Haskell has a predefined infix operator `||` that serves the same purpose as `disj`. Instead of the above, we could have used the following:

```
LOLA1> filter (\ x -> (r A x) || (r B x)) entities
[A,B,C]
```

Similarly, Haskell has a predefined infix operator `&&` for conjunction.

The Language of Typed Logic and Its Semantics

Assume that we have constants and variables available for all types in the type hierarchy. Then the language of typed logic over these is defined as follows.

$$\begin{array}{rcl}
\textbf{type} & ::= & e \mid t \mid (\textbf{type} \to \textbf{type}) \\
\textbf{expression} & ::= & \textbf{constant}_{\textbf{type}} \mid \textbf{variable}_{\textbf{type}} \\
 & \mid & (\lambda\ \textbf{variable}_{\textbf{type}_1}.\textbf{expression}_{\textbf{type}_2})_{\textbf{type}_1 \to \textbf{type}_2} \\
 & \mid & (\textbf{expression}_{\textbf{type}_1 \to \textbf{type}_2}\ \textbf{expression}_{\textbf{type}_1})_{\textbf{type}_2}
\end{array}$$

For an expression of the form $(E_1 E_2)$ to be *welltyped* the types have to match. The type of the resulting expression is fully determined by the types of the components. Similarly, the type of a lambda expression $(\lambda v.E)$ is fully determined by the types of v and E.

Often we leave out the type information. The definition of the language then looks like this:

$$\begin{array}{rcl}
\textbf{expression} & ::= & \textbf{constant} \\
 & | & \textbf{variable} \\
 & | & (\lambda\ \textbf{variable}.\textbf{expression}) \\
 & | & (\textbf{expression expression})
\end{array}$$

Models for Typed Logic

A model M for a typed logic over e, t consists of a domain $\mathrm{dom}\,(M) = D_e$ together with an interpretation function $\mathrm{int}\,(M) = I$ which maps every constant of the language to a function of the appropriate type in the domain hierarchy based on D_e.

A variable assignment s for typed logic maps every variable of the language to a function of the appropriate type in the domain hierarchy.

The semantics for the language is given by defining a function $[\![.]\!]_s^M$ which maps every expression of the language to a function of the appropriate type.

$$[\![\textbf{constant}]\!]_s^M = I(\textbf{constant}).$$

$$[\![\textbf{variable}]\!]_s^M = s(\textbf{variable}).$$

$$[\![(\lambda v_{T_1}.E_{T_2})]\!]_s^M = h$$

where h is the function given by $h : d \in D_{T_1} \mapsto [\![E]\!]_{s(v|d)}^M \in D_{T_2}$.

$$[\![(E_1 E_2)]\!]_s^M = [\![E_1]\!]_s^M([\![E_2]\!]_s^M).$$

Assume that $(((Gc)b)a)$ expresses that a gives b to c.

What do the following expressions say:

1. $(\lambda x.(((Gc)b)x))$.

   giving b to c

2. $(\lambda x.(((Gc)x)a))$.

   being a present of a to c

3. $(\lambda x.(((Gx)b)a))$.

   receiving b from a

4. $(\lambda x.(((Gx)b)x))$.

   giving b to oneself.

Assume that $(((Gc)b)a)$ expresses that a gives b to c. What do the following expressions say:

1. $(\lambda x.(\lambda y.(((Gx)b)y)))$.

   giving b

2. $(\lambda x.(\lambda y.(((Gy)b)x)))$.

   receiving b.

Logical Constants of Predicate Logic

The logical constants of predicate logic can be viewed as constants of typed logic, as follows. $\neg$ is a constant of type $t \rightarrow t$ with the following interpretation.

- $[\![\neg]\!] = h$, where h is the function in $\{0,1\}^{\{0,1\}}$ which maps 0 to 1 and vice versa.

$\wedge$ and $\vee$ are constants of type $t \rightarrow t \rightarrow t$ with the following interpretations.

- $[\![\wedge]\!] = h$, where h is the function in $(\{0,1\}^{\{0,1\}})^{\{0,1\}}$ which maps 1 to $\{(1,1),(0,0)\}$ and 0 to $\{(1,0),(0,0)\}$.
- $[\![\vee]\!] = h$, where h is the function in $(\{0,1\}^{\{0,1\}})^{\{0,1\}}$ which maps 1 to $\{(1,1),(0,1)\}$ and 0 to $\{(1,1),(0,0)\}$.

```
not          :: Bool -> Bool
not True      = False
not False     = True
```

```
(&&)  :: Bool -> Bool -> Bool
False && x   = False
True  && x   = x
```

```
(||)  :: Bool -> Bool -> Bool
False || x   = x
True  || x   = True
```

Quantifiers of Predicate Logic

The quantifiers $\exists$ and $\forall$ are constants of type $(e \rightarrow t) \rightarrow t$, with the following interpretations.

- $[\![\forall]\!] = h$, where h is the function in $\{0,1\}^{D_{e\rightarrow t}}$ which maps the function that characterizes D_e to 1 and every other characteristic function to 0.
- $[\![\exists]\!] = h$, where h is the function in $\{0,1\}^{D_{e\rightarrow t}}$ which maps the function that characterizes $\emptyset$ to 0 and every other characteristic function to 1.

Second Order and Polymorphic Quantification

It is possible to add constants for quantification over different types.

E.g., to express second order quantification (i.e., quantification over properties of things), one would need quantifier constants of type

$$((e \rightarrow t) \rightarrow t) \rightarrow t.$$

The general (polymorphic) type of quantification is

$$(T \rightarrow t) \rightarrow t.$$

Haskell implementation of quantification:

```
any, all          :: (a -> Bool) -> [a] -> Bool
any p              = or   . map p
all p              = and . map p
```

Definition of `forall` and `exists` for bounded enumerable domains, in terms of these:

```
forall, exists ::
  (Enum a, Bounded a) => (a -> Bool) -> Bool
forall = \ p -> all p [minBound..maxBound]
exists = \ p -> any p [minBound..maxBound]
```

What is the type of binary generalized quantifiers such as

$$Q_\forall(\lambda x.Px)(\lambda x.Qx) \quad \text{('all P are Q')}$$

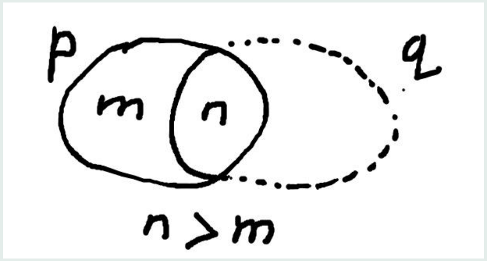
$$Q_\exists(\lambda x.Px)(\lambda x.Qx) \quad \text{('some P are Q')}$$

$$Q_M(\lambda x.Px)(\lambda x.Qx) \quad \text{('most P are Q')}$$

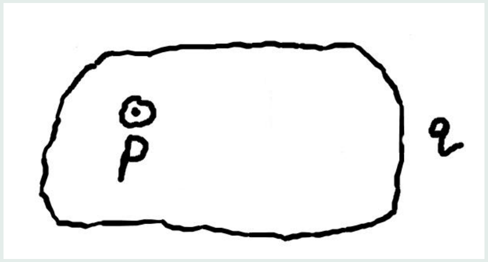
$$(e \to t) \to (e \to t) \to t.$$

Implementation of binary generalized quantifiers:

```
every, some, no ::
  (Entity -> Bool) -> (Entity -> Bool) -> Bool
every = \ p q -> all q (filter p entities)
some  = \ p q -> any q (filter p entities)
no    = \ p q -> not (some p q)
```



```
most ::
  (Entity -> Bool) -> (Entity -> Bool) -> Bool
most = \ p q ->
 length (filter (\ x -> p x && q x) entities)
 >
 length (filter (\ x -> p x && not (q x)) entities)
```



'The P is Q' is true just in case

1. there is exactly one P,
2. that P is Q.

```haskell
singleton :: [a] -> Bool
singleton [x] = True
singleton  _  = False
```

```haskell
the ::
  (Entity -> Bool) -> (Entity -> Bool) -> Bool
the = \ p q ->
   singleton (filter p entities)
   &&
   q (head (filter p entities))
```

Typed Logic and Predicate Logic

With the constants $\neg, \wedge, \vee, \forall, \exists$ added, the language of typed logic contains ordinary predicate logic as a proper fragment.

Here are some example expressions of typed logic, with their predicate logical counterparts:

$(\neg(Px))$	$\neg Px$
$((\wedge(Px))(Qy))$	$Px \wedge Qy$
$(\forall(\lambda x.Px))$	$\forall x Px$
$(\forall(\lambda x.(\exists(\lambda y.((Ry)x)))))$	$\forall x \exists y Rxy$

Note the order of the arguments in $((Ry)x)$ versus Rxy.

Curry and Uncurry

The conversion of a function of type `(a,b) -> c` to one of type `a -> b -> c` is called *currying* (named after Haskell Curry).

`curry` is predefined in Haskell:

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

The conversion of a function of type `a -> b -> c` to one of type `(a,b) -> c` is called *uncurrying*,

```
uncurry             :: (a -> b -> c) -> ((a,b) -> c)
uncurry f p          = f (fst p) (snd p)
```

```
LOLA1> r B C
True
LOLA1> (uncurry r) (B,C)
True
```

As we also want the arguments in the other order, what we need is:

1. first flip the arguments,
2. next uncurry the function.

```
inv2     :: (a -> b -> c) -> ((b,a) -> c)
inv2 = uncurry . flip
```

```
LOLA1> inv2 r (C,B)
True
```

For the converse operation, we combine `curry` with `flip`, as follows: `flip . curry`.

```
LOLA1> (flip . curry . inv2) r B C
True
```

Conversion for Three Placed Relations

For three-placed relations, we need to convert from and too triples, and for that appropriate definitions are needed for selecting the first, second or third element from a triple.

```
e1            :: (a,b,c) -> a
e1 (x,y,z)     = x
e2            :: (a,b,c) -> b
e2 (x,y,z)     = y
e3            :: (a,b,c) -> c
e3 (x,y,z)     = z

inv3        :: (a -> b -> c -> d) -> ((c,b,a) -> d)
inv3 f t     = f (e3 t) (e2 t) (e1 t)
```

Here is a definition of a three-placed relation in Haskell:

```
g :: Entity -> Entity -> Entity -> Bool
g D x y = r x y
g E x y = not (r x y)
g _ _ _ = False
```

This gives:

```
LOLA1> g E B C
False
LOLA1> g E C B
True
LOLA1> inv3 g (B,C,E)
True
LOLA1>
```

Assume R is a typed logical constant of type $e \to (e \to t)$ which is interpreted as the two-placed relation of respect between individuals. Then $((Rb)a)$ expresses that a respects b. In abbreviated notation this becomes $R(a,b)$. Now $\lambda x.\exists y R(x,y)$ is abbreviated notation for 'respecting someone', and $\lambda x.\exists y R(x,y)$ for 'being respected by someone'.

If $G(a,b,c)$ is shorthand for 'a gives b to c', then $\lambda x.\exists y \exists z G(x,y,z)$ expresses 'giving something to someone', and $\lambda x.\exists y G(x,b,y)$ expresses 'giving b to someone'. Finally, $\lambda x.\exists y G(y,b,x)$ expresses 'receiving b from someone'.

Assume $G(x,y,z)$, which is short for $(((Gz)y)x)$, means that x gives y to z. Use this to find a typed logic expression for 'receiving something from someone'.

$$\lambda x.\exists y \exists z G(y,z,x).$$

Suppose we want to translate ‘Anne gave Claire the book’, which has syntactic structure

$$[_S[_{NP}\textit{Anne}\ ][_{VP}[_{TV}[_{DTV}\textit{gave}][_{NP}\textit{Claire}\ ]][_{NP}\textit{the book}\ ]]]$$

in a compositional way, using $\lambda zyx.G(x,y,z)$ as translation for ‘give’. Translating all the combinations of phrases as function argument combination, we arrive at the following translation of the sentence.

1 $(((\lambda zyx.G(x,y,z)c)b)a)$.

This does indeed express what we want, but in a rather roundabout way. We would like to *reduce* this expression to $G(a,b,c)$.

Reducing Expressions of Typed Logic

To reduce expression (1) to its simplest form, three steps of so-called β conversion are needed. During β conversion of an expression consisting of a functor expression $\lambda v.E$ followed by an argument expression A, basically the following happens:

The prefix $\lambda v.$ is removed, the argument expression A is removed, and finally the argument expression A is *substituted* in E for all free occurrences of v. The free occurrences of v in E are precisely the occurrences which were bound by λv in $\lambda v.E$.

Here is the *proviso*. In some cases, the substitution process described above cannot be applied without further ado, because it will result in unintended capturing of variables within the argument expression A.

Consider expression (2):

2 $((\lambda x.(\lambda y.((Ry)x)))y)$.

In this expression, y is bound in the functional part $(\lambda x.(\lambda y.((Ry)x)))$ but free in the argument part y. Reducing (2) by β conversion according to the recipe given above would result in $(\lambda y.((Ry)y))$, with capture of the argument y at the place where it is substituted for x.

This problem can be avoided by performing β conversion on an *alphabetic variant* of the original expression, say on (3).

3 $((\lambda x.(\lambda z.((Rz)x)))y)$.

Another example where α conversion (i.e., switching to an alphabetic variant) is necessary before β conversion to prevent unintended capture of free variables is the expression (4).

4 $((\lambda p.\forall x((Ax) \leftrightarrow p))(Bx)).$

In (4), p is a variable of type t, and x one of type e. Variable x is bound inside the functional part $(\lambda p.\forall x((Ax) \leftrightarrow p))$ but free in the argument part (Bx). Substituting (Bx) for p in the function expression would cause x to be captured, with failure to preserve the original meaning.

Again, the problem is avoided if β conversion is performed on an alphabetic variant of the original expression, say on (5).

5 $((\lambda p.\forall z((Az) \leftrightarrow p))(Bx)).$

Performing β reduction on (5) yields $\forall z.((Ax) \leftrightarrow (Bx))$, with the argument of B still free, as it should be.

Freedom, Bondage, Substitution

Variable v is free in expression E if the following holds (we use $\approx$ for the relation of being syntactically identical, i.e. for being the same expression):

- $v \approx E$,
- $E \approx (E_1 E_2)$, and v is free in E_1 or v is free in E_2,
- $E \approx (\lambda x.E_1)$, and $v \not\approx x$, and v is free in E_1.

Examples of free occurrences of x:

$(\lambda x.(Px)), ((\lambda x.(Px))x), (\lambda x.((Rx)x)), ((\lambda x.((Rx)x))x), ((\lambda y.((Rx)y))x)$.

Which occurrences of x are free in $((\lambda y.\exists x((Rx)y))x)$?

Bear in mind that $\exists x((Rx)y)$ is shorthand for $(\exists(\lambda x.((Rx)y)))$.

$$((\lambda y.\exists x((Rx)y))x).$$

Substitution of y for x in $(\lambda y.(Px))$:

$$(\lambda y.(Px))[x := y].$$

The function $(\lambda y.(Px))$ is the function yielding (Px) for any argument, i.e., the constant (Px) function.

It is easy to get this substitution wrong:

$$(\lambda y.(Px))[x := y] \approx (\lambda y.(Px)[x := y]) \approx (\lambda y.(Py)).$$

This is a completely different function, namely the function that assigns to argument a the result of applying P to a.

By doing an appropriate renaming, we get the correct result:

$$(\lambda y.(Px))[x := y] \approx (\lambda z.(Px)[y := z][x := y])$$

$$\approx (\lambda z.(Px)[x := y]) \approx (\lambda z.(Py)).$$

Substitution of s for free occurrences of v in E, with notation $E[v := s]$.

- If $E \approx v$ then $E[v := s] \approx s$,
  if $E \approx x \not\approx v$ (i.e., E is a variable different from v), then $E[x := s] \approx x$,
  if $E \approx c$ (i.e., E is a constant), then $E[x := s] \approx c$,
- if $E \approx (E_1 E_2)$ then $E[v := s] \approx (E_1[v := s]\ E_2[v := s])$,
- if $E \approx (\lambda x.E_1)$, then
  - if $v \approx x$ then $E[v := s] \approx E$,
  - if $v \not\approx x$ then there are two cases:
    1. if x is not free in s or v is not free in E then $E[v := s] \approx (\lambda x.E_1[v := s])$,
    2. if x is free in s and v is free in E then $E[v := s] \approx (\lambda y.E_1[x := y][v := s])$, for some y which is not free in s and not free in E_1.

Reduction

Reduction comes in three flavours: β reduction, α reduction and η reduction, with corresponding arrows $\xrightarrow{\beta}$, $\xrightarrow{\alpha}$ and $\xrightarrow{\eta}$.

Beta reduction: $((\lambda v.E)s) \xrightarrow{\beta} E[v := s]$.

Condition: v and s are of the same type (otherwise the expression to be reduced is not welltyped).

Alpha reduction: $(\lambda v.E) \xrightarrow{\alpha} (\lambda x.E[v := x])$.

Conditions: v and x are of the same type, and x is not free in E.

Eta reduction: $((\lambda v.E)v) \xrightarrow{\eta} E$.

The ‘real work’ takes place during β reduction. The α reduction rule serves only to state in an explicit fashion that λ calculations are insensitive to switches to alphabetic variants. Whether one uses λx to bind occurrences of x or λy to bind occurrences of y is immaterial, just like it is immaterial in the case of predicate logic whether one writes $\forall x P x$ or $\forall y P y$.

The η reduction rule makes a principle explicit that we have used implicitly all the time: if (Pj) expresses that John is present, then both P and $(\lambda x.(Px))$ express the property of being present. This is so because $((\lambda x.(Px))x) \xrightarrow{\eta} (Px)$, so P and $(\lambda x.(Px))$ give the same result when applied to argument x, i.e., they express the same function.

Applying β reduction to

$$(((\lambda zyx.G(x,y,z)c)b)a),$$

or in unabbreviated notation

$$((((\lambda z.(\lambda y.(\lambda x.(((Gz)y)x))))c)b)a),$$

gives:

$$\begin{aligned}
((((\lambda z.(\lambda y.(\lambda x.(((Gz)y)x))))c)b)a) &\xrightarrow{\beta} (((\lambda y.(\lambda x.(((Gc)y)x)))b)a) \\
&\xrightarrow{\beta} ((\lambda x.(((Gc)b)x))a) \\
&\xrightarrow{\beta} (((Gc)b)a).
\end{aligned}$$

To be fully precise we have to state explicitly that expressions can be reduced ‘in context’. The following principles express this:

$$\frac{E \xrightarrow{\beta} E'}{(F\,E) \xrightarrow{\beta} (F\,E')} \qquad \frac{E \xrightarrow{\beta} E'}{(E\,F) \xrightarrow{\beta} (E'\,F)}$$

$$\frac{E \xrightarrow{\beta} E'}{(\lambda v.E) \xrightarrow{\beta} (\lambda v.E')}$$

Here F is assumed to have the appropriate type.

These principles allow β reductions at arbitrary depth within expressions.

Reduce the following expressions to their simplest forms:

1. $((\lambda Y.(\lambda x.(Yx)))P)$.

   $\lambda x.(Px)$.

2. $(((\lambda Y.(\lambda x.(Yx)))P)y)$.

   (Py).

3. $((\lambda P.(\lambda Q.\exists x(Px \wedge Qx)))A)$.

   $(\lambda Q.\exists x(Ax \wedge Qx))$.

4. $(((\lambda P.(\lambda Q.\exists x(Px \wedge Qx)))A)B)$.

   $\exists x(Ax \wedge Bx)$.

5. $((\lambda P.(\lambda Q.\forall x(Px \Rightarrow Qx)))(\lambda y.((\lambda x.((Ry)x))j)))$.

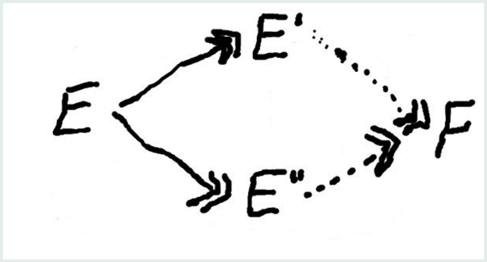
   $(\lambda Q.\forall x((\lambda z.((Rj)z)x) \Rightarrow Qx))$

   $(\lambda Q.\forall x(((Rj)x) \Rightarrow Qx))$.

Confluence Property

In the statement of the following property, we write $E \twoheadrightarrow E'$ for E *reduces in a number of* α, β, η *steps to E'*.

Confluence property (or: Church-Rosser property): For all expressions E, E_1, E_2 of typed logic: if $E \twoheadrightarrow E_1$ and $E \twoheadrightarrow E_2$ then there is an expression F with $E_1 \twoheadrightarrow F$ and $E_2 \twoheadrightarrow F$.



Normal Form Property

An expression of the form $((\lambda v.E)s)$ is called a β-*redex* (for: β reducible expression). $E[v := s]$ is called the *contractum* of $((\lambda v.E)s)$. An expression that does not contain any redexes is called a *normal form*.

Normal form property: Every expression of typed logic can be reduced to a normal form.

Combining the confluence property and the normal form property we get that the normal forms of an expression E are identical modulo α conversion. That is to say, all normal forms of E are alphabetic variants of one another.

The normal form property holds thanks to the restrictions imposed by the typing discipline. Untyped lambda calculus lacks this property. In untyped lambda calculus it is allowed to apply expressions to themselves. In typed lambda calculus this is forbidden, because (xx) cannot be consistently typed.

In untyped lambda calculus, expressions like $(\lambda x.(xx))$ are wellformed. It is easy to see that

$$((\lambda x.(xx))(\lambda x.(xx)))$$

does not have a normal form.

Misleading Form and Logical Form

The metamorphosis of β conversion is relevant for an enlightened view on the historical 'misleading form thesis' for natural language.

The predicate logical translations of natural language sentences with quantified expressions did not seem to follow the linguistic structure. In the logical translations, the quantified expressions seemed to have disappeared.

Using expressions of typed logic it is quite simple to analyse *John smiled* and *No-one smiled* along the same lines: The subject NPs get the translations:

$$\lambda P.Pj$$

and

$$\lambda P.\neg\exists x.((\textsf{person}\ x) \wedge (Px)).$$

Representing a Model for Predicate Logic in Haskell

All we need to specify a first order model in Haskell is a domain of entities and suitable interpretations of proper names and predicates. The domain of entities was given above as `Entity`.

```
module Model where

import Domain
```

Interpretation for proper names:

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

```
man,boy,woman,person,thing,house :: Entity -> Bool
man x    = x == B || x == D || x == J
woman x  = x == A || x == C || x == M
boy x    = x == J
person x = man x || woman x
thing x  = not (person x)
house x  = x == H
cat x    = x == K
mouse x  = x == I
```

```
laugh,cry,curse,smile,old,young :: Entity -> Bool
laugh x  = x == M || x == C || x == B
        || x == D || x == J
cry   x  = x == B || x == D
curse x  = x == J || x == A
smile x  = x == M || x == B || x == I || x == K
young x  = x == J
old x    = x == B || x == D
```

Meanings for two-placed predicates, represented as `(Entity,Entity) -> Bool`.

```
love,respect,hate,own :: (Entity,Entity) -> Bool
love (x,y)    = (x == B && (y == M || y == A))
                || (woman x && (y == J || y == B))
respect (x,y) = person x && person y
                || x == F || x == I
hate (x,y)    = ((x == B || x == J) && thing y)
                || (x == I && y == K)
own (x,y)     =  (x ==A && y == E)
                || (x == M && (y == K || y == H))
```

Montague Grammar: Datastructures for Syntax

```
data S = S 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
     deriving (Eq,Show)

data CN = Man   | Woman | Boy | Person
        | Thing | House | Cat | Mouse
     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)
```

Semantic Interpretation

We define for every syntactic category an interpretation function of the appropriate type.

Sentences:

```
intS :: S -> Bool
intS (S np vp) = (intNP np) (intVP vp)
```

Noun Phrases:

```
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)
```

For the interpretation of verb phrases we invoke the information encoded in our first order model.

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

For the interpretation of complex VPs, we have to find a way to make reference to the property of ‘standing into the TV relation to the subject of the sentence’.

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

TVs: link up with model information.

```haskell
intTV :: TV -> Entity -> Entity -> Bool
intTV Loved     = (flip . curry) love
intTV Respected = (flip . curry) respect
intTV Hated     = (flip . curry) hate
intTV Owned     = (flip . curry) own
```

The interpreation of CNs is similar to that of VPs.

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

Determiners: use the logical constants defined above.

```
intDET :: DET ->
  (Entity -> Bool) -> (Entity -> Bool) -> Bool
intDET Every = every
intDET Some  = some
intDET No    = no
intDET The   = the
```

Interpretation of relativised common nouns of the form *That CN VP*: check whether an entity has both the CN and the VP property:

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

Interpretation of relativised common nouns of the form *That CN NP TV*: check whether an entity has both the CN property as the property of being the object of *NP TV*.

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

Testing it Out

```
LOLA1> intS (S (NP1 The Boy) Smiled)
False

LOLA1> intS (S (NP1 The Boy) Laughed)
True

LOLA1> intS (S (NP1 Some Man) Laughed)
True

LOLA1> intS (S (NP1 No Man) Laughed)
False

LOLA1> intS (S (NP1 Some Man) (VP1 Loved (NP1 Some Woman)))
True
```

Further Issues

- Syntax: if there is time.
- Quantifying-in: application of NP denotations to 'PRO-abstractions'.
- Intensionality: add a type w for worlds, and build the type hierarchy over t, e, w.
- Flexible typing
- Anaphoric linking: classical variable binding is not enough ...

Next Lecture: The Dynamic Turn

References

[Bir98] R. Bird. *Introduction to Functional Programming Using Haskell*. Prentice Hall, 1998.

[DvE02] K. Doets and J. van Eijck. Reasoning, computation and representation using Haskell. Draft Textbook. Available from `www.cwi.nl/~jve/RCRH.ps.gz`, 2002.

[HFP96] P. Hudak, J. Fasel, and J. Peterson. A gentle introduction to Haskell. Technical report, Yale University, 1996. Available from the Haskell homepage: `http://www.haskell.org`.

[Hin97] J. Roger Hindley. *Basic Simple Type Theory*. Cambridge University Press, 1997.

[JH$^+$99] S. Peyton Jones, J. Hughes, et al. Report on the programming language Haskell 98. Available from the Haskell homepage: `http://www.haskell.org`, 1999.

[Mon73] R. Montague. The proper treatment of quantification in ordinary English. In J. Hintikka e.a., editor, *Approaches to Natural Language*, pages 221–242. Reidel, 1973.