My Report

Me

Tuesday 18th March, 2025

Abstract

We give a toy example of a report in *literate programming* style. The main advantage of this is that source code and documentation can be written and presented next to each other. We use the listings package to typeset Haskell source code nicely.

Contents

1 $\mathcal{KL}$: Syntax and Semantics

2 Syntax of $\mathcal{KL}$

The syntax of the language $\mathcal{KL}$ is described in **Lokb** and inspired by Levesque's work (**levesque1981**). The SyntaxKL module establishes the foundation for $\mathcal{KL}$'s syntax, defining the alphabet and grammar used in subsequent semantic evaluation.

```
module SyntaxKL where
```

Symbols of $\mathcal{KL}$

The expressions of $\mathcal{KL}$ are constituted by sequences of symbols drawn from the following two sets (cf. **levesque1981**): Firstly, the *logical symbols*, which consist of the logical connectives and quantifiers $\exists, \vee, \neg$, as well as punctuation and parentheses. Furthermore, it compromises a countably infinite supply of first-order variables denoted by the set $\{x, y, z, \ldots\}$, a countably infinite supply of standard names, represented by the set $\{\#1, \#2, \ldots\}$, and the equality symbol $=$. The *non-logical symbols* comprise predicate symbols of any arity $\{P, Q, R, \ldots\}$, which are intended to represent domain-specific properties and relations, and function symbols of any arity, which are used to denote mappings from individuals to individuals (**Lokb**, p.22).
n this implementation, standard names are represented as strings (e.g., "n1", "n2") via the StdName type, and variables are similarly encoded as strings (e.g., "x", "y") with the Variable type, ensuring that we have a distinct yet infinite supplies of each.

```
-- Represents a standard name (e.g., "n1") from the infinite domain N
newtype StdName = StdName String deriving (Eq, Ord, Show)

-- Represents a first-order variable (e.g., "x")
newtype Variable = Var String deriving (Eq, Ord, Show)
```

Terms and Atoms

Terms in $\mathcal{KL}$ are the building blocks of expressions, consisting of variables, standard names, or function applications. Atomic propositions (atoms) are formed by applying predicate symbols to lists of terms. To distinguish primitive terms (those that contain no variable and only a single function symbol) and primitive atoms (those atoms that contain no variables and only standard names as terms) for semantic evaluation, we also define PrimitiveTerm and PrimitiveAtom.

```
-- Defines terms: variables, standard names, or function applications
data Term = VarTerm Variable   -- A variable (e.g., "x")
          | StdNameTerm StdName   -- A standard name (e.g., "n1")
          | FuncApp String [Term]   -- Function application (e.g., "Teacher" ("x"))
          deriving (Eq, Ord, Show)

-- Terms with no variables and only a single function symbol
data PrimitiveTerm = PStdNameTerm StdName   -- e.g., "n1"
                   | PFuncApp String [StdName]
  deriving (Eq, Ord, Show)

-- Define Atoms as predicates applied to terms
data Atom = Pred String [Term]  --e.g. "Teach" ("n1", "n2")
  deriving (Eq, Ord, Show)

-- Atoms with only standard names as terms
data PrimitiveAtom = PPred String [StdName]
  deriving (Eq, Ord)
```

Formulas

$\mathcal{KL}$-formulas are constructed recursively from atoms, equality, and logical operators. The

Formula type includes atomic formulas, equality between terms, negation, disjunction, existential quantification, and the knowledge operator K. Additional connectives like universal quantification ($\forall$), implication ($\rightarrow$), and biconditional ($\leftrightarrow$) are defined as derived forms for convenience.

```
--Defines KL-formulas with logical and epistemic constructs
data Formula = Atom Atom                -- Predicate (e.g. Teach(x, "n1"))
             | Equal Term Term          --Equality (e.g., x = "n1")
             | Not Formula              -- Negation
             | Or Formula Formula       -- Disjunction
             | Exists Variable Formula -- Existential (e.g., exists x (Teach x "sue"))
             | K Formula                -- Knowledge Operator (e.g., K (Teach "ted" "sue"))
             deriving (Eq, Show)

-- Universal quantifier as derived form
for_all :: Variable -> Formula -> Formula
for_all x f = Not (Exists x (Not f))

-- Implication as derived form
implies :: Formula -> Formula -> Formula
implies f1 f2 = Or (Not f1) f2

-- Biconditional as derived form
iff :: Formula -> Formula -> Formula
iff f1 f2 = Or (Not (Or f1 f2)) (Or (Not f1) f2)
```

We can now use this implementation of $\mathcal{KL}$'s syntax to implement the semantics.

3 Wrapping it up in an exectuable

We will now use the library form Section **??** in a program.

```
module Main where

import Basics

main :: IO ()
main = do
  putStrLn "Hello!"
  print somenumbers
  print (map funnyfunction somenumbers)
  myrandomnumbers <- randomnumbers
  print myrandomnumbers
  print (map funnyfunction myrandomnumbers)
  putStrLn "GoodBye"
```

We can run this program with the commands:

```
stack build
stack exec myprogram
```

The output of the program is something like this:

```
Hello!
[1,2,3,4,5,6,7,8,9,10]
[100,100,300,300,500,500,700,700,900,900]
[1,3,0,1,1,2,8,0,6,4]
[100,300,42,100,100,100,700,42,500,300]
GoodBye
```

references.bib