CS 3EA3: Example Haskell Code for Propositional Logic

Wolfram Kahl

September 15, 2009

1 Introduction

This is a LaTeX document binding together some Haskell modules (currently only one). It expects to by processed by `lhs2TeX` first, and then by LaTeX; for directly producing PDF output, you would issue the following commands:

```
lhs2TeX --poly PropDoc.lhs > PropDoc.ltx
pdflatex PropDoc.ltx
```

2 Propositional Formulae

A datatype for simple propositional logic formulae, with strings used for propositional variables.

```
data Prop
   = Var String
   | Negation Prop
   | BinOp Op Prop Prop
```

To allow more generalised treatment of the binary operators, we included only a single alternative for all of them, and provide a separate datatype to contain just the binary operators:

```
data Op = And | Or | Implies | Equiv
```

Then the choice of operator symbols can be encapsulated in the function *showOp* that is completely independent of the *Prop* context:

```
showOp :: Op → String
showOp And     = "&"
showOp Or      = "|"
showOp Implies = "=>"
showOp Equiv   = "<=>"
```

We want the standard prelude function *show*, when applied to *Op* values, to be implemented by *showOp*, and achieve this by adding an **instance** declaration for the type class *Show* which provides *show*:

```
instance Show Op where show = showOp
```

Converting propositions to strings is of course recursive; in Haskell, recursion does not have to be declared — the whole module can consist of mutually recursive definitions.

```
showProp :: Prop → String
showProp (Var s) = s
showProp (Negation p) = "~" ++ showProp p
showProp (BinOp op p q) = paren (showProp p ++ space (showOp op) ++ showProp q)
```

In the *BinOp* case in the definition of *showProp*, we used uses the following two auxiliary functions — note that in Haskell, definitions can be presented in arbitrary sequence.

```
space s = "  " ++ s ++ "  "
paren s = "(" ++ s ++ ")"
```

Again, we can install *showProp* to be used as *show* on *Prop* values:

```
instance Show Prop where show = showProp
```

To make formula construction in programs and in GHCi easier, we define infix operators for the binary junctors, let them all associate to the right, and impose the same precedence ordering as Z:

```
lnot p = Negation p
p &&& q = BinOp And p q
p ||| q = BinOp Or p q
p ==> q = BinOp Implies p q
p <=> q = BinOp Equiv p q
infixr 7 &&&
infixr 6 |||
infixr 5 <=>
infixr 4 ==>
```

Another step that helps with example formula construction is providing simple names for the some frequently-used propositional variables:

```
p = Var "p"
q = Var "q"
r = Var "r"
s = Var "s"
```

With all this, defining example propositions becomes easy:

```
prop1 = p &&& q ==> p ||| q
```