Programming Languages and Computation

# Problem Sheet 10: Gödel numbering

We will use the techniques developed in the lectures to develop an encoding of the abstract syntax of While programs as natural numbers. The result will be that While programs will be able to manipulate While code themselves! This might sound circular or impossible at first. But it is not: if While programs are encoded as natural numbers, then any language that is able to compute with natural numbers is able to manipulate While syntax (or—better—the *reflection* of that manipulation).

As the natural numbers we will use are going to get very large, we will be using the `Integer` data type, which supports arbitrary-precision integers.

We will be working with the file while/src/Goedel.hs in the labcode repository.

To begin we import the definition of the While AST. We also import the parser function, which will be useful in generating some examples:

```
import WhileAST
import Parser (parse)
```

## Encoding Booleans

First, we will implement the section-retraction pair that encodes Boolean values as natural numbers.

* 1. Complete the definitions of the functions

```
encodeBool :: Bool → Integer
decodeBool :: Integer → Bool
```

Think carefully about what the retraction `decodeBool` should do. For mathematics we prefer it to be total. But for implementation purposes it may be far more practical to make it throw an error whenever it attempts to decode something that is *not* the encoding of a Boolean value.

Solution

```haskell
encodeBool :: Bool → Integer
encodeBool True = 0
encodeBool False = 1

decodeBool :: Integer → Bool
decodeBool 0 = True
decodeBool 1 = False
decodeBool _ = error "number does not represent a Boolean"
```

---

## Encoding pairs

Second, we need to implement a pairing function.

** 2. Complete the definitions of the functions

```haskell
pair :: Integer → Integer → Integer
unpair :: Integer → (Integer, Integer)
```

Your definitions should satisfy the section-retraction property that, roughly, for any `x, y :: Integer` we have `unpair (pair x y) = (x, y)`.
There are many ways to implement a pairing function. Of course you can go ahead and implement the one in the lecture. However, that involves computing $2^m$ for very large $m$. This will result in a very slow encoding which might take an amount of time comparable to the chronology of the universe when encoding any program that is longer than a few instructions. Instead, I recommend that you implement the Cantor variant of the Hopcroft-Ullman pairing function. The beauty of Haskell is that you should be able to translate the definitions into code with minimal effort.
(Hint: you may find the function fromInteger useful.)

Solution

```
delta :: Integer → Integer
delta x = (x * (x + 1)) `div` 2

pairHU :: Integer → Integer → Integer
pairHU i j = delta (i + j - 2) + i

unpairHU :: Integer → (Integer, Integer)
unpairHU h = (i, j)
 where
  c = floor (sqrt (2 * fromInteger h) - (1/2))
  i = h - delta c
  j = c - i + 2

pair :: Integer → Integer → Integer
pair i j = pairHU (i + 1) (j + 1) - 1

unpair :: Integer → (Integer, Integer)
unpair h = let (x, y) = unpairHU (h + 1) in (x - 1, y - 1)
```

---

## Encoding binary trees

As a warm-up for encoding While programs, we first implement the encoding of binary trees given in the lecture. We define the binary tree data type by

```
data BTree = Empty | Fork Integer BTree BTree
 deriving (Eq, Show)
```

To generate some test cases with minimal effort I have provided the definition of a function

```
insert :: Integer → BTree → Btree
```

which inserts an integer into a binary tree, so that the Binary Search Tree (BST) invariant is maintained. One can then use `foldr` to `insert` a list of numbers into an empty tree, therefore generating a BST. If the list is given in a relatively unsorted condition, this tree will also be somewhat balanced.

** 3. Complete the definition of the functions

```
encodeBTree :: BTree → Integer
decodeBTree :: Integer → BTree
```

so that `decodeBTree (encodeBTree t) = t` for all `t :: BTree`.

Solution

```
encodeBTree :: BTree → Integer
encodeBTree Empty = 0
encodeBTree (Fork n t1 t2) = 1 + pair n (pair (encodeBTree t1) (encodeBTree t2))

decodeBTree :: Integer → BTree
decodeBTree 0 = Empty
decodeBTree n =
  let (k, t)   = unpair (n - 1)
      (t1, t2) = unpair t
  in Fork k (decodeBTree t1) (decodeBTree t2)
```

---

## Aside: encoding variables

Before attempting the encode any While programs themselves we have to find a way to encode variable names. To achieve that one may, for example, use the `chr` and `ord` functions of `Data.Char` to convert individual characters as numbers, and then represent variable names as lists of characters. However, any such method is bound to yield large numbers, thereby making our numbering cosmically slow.

Instead, we will limit the language to variables of the form `x1`, `x2`, .... We will then encode such variables by the number that follows the character `x`. The code that achieves this is

```
encodeVar :: String → Integer
encodeVar ('x':s) = read s
encodeVar _       = error "not a valid variable"

decodeVar :: Integer → String
decodeVar n = "x" ++ show n
```

## Encoding arithmetic expressions

To encode arithmetic expressions we use the same trick as in binary trees: we exploit Euclidean division. Dividing by a positive integer $m \in \mathbb{N}^+$ gives us that any positive integer $n \in \mathbb{N}^+$ can be uniquely written as $n = q * m + r$ where $0 \leq r < m$. The number $q$ is called the *quotient*, and $r$ is the *remainder* of the division.

We pick the divisor depending on the number of kinds of AST nodes we are trying to encode. In our case, we let $m = 5$, and define the function

```
encodeAExpr :: AExpr → Integer
encodeAExpr (EVar xs) = 5 * encodeVar xs
encodeAExpr (EInt n)  = 1 + 5 * n
encodeAExpr (EBinOp AAdd a1 a2) = 2 + 5 * pair (encodeAExpr a1) (encodeAExpr a2)
encodeAExpr (EBinOp ASub a1 a2) = 3 + 5 * pair (encodeAExpr a1) (encodeAExpr a2)
encodeAExpr (EBinOp AMul a1 a2) = 4 + 5 * pair (encodeAExpr a1) (encodeAExpr a2)
```

Notice that this function does a nested pattern-match: first, it matches on the AST of arithmetic expressions, giving three cases (variables, integers, binary operations). If the AST node is that of a binary operation, it pattern-matches on the operation itself (addition, subtraction, and multiplication). This gives a total of 5 different 'kinds' of AST node—hence the choice of $m = 5$.

To decode an arithmetic expression we must perform Euclidean division by $m = 5$. The remainder $(0, 1, 2, 3, 4)$ tells us which of the five kinds of expression it is. Following that, we must decode the quotient to obtain the values of the subtrees.

*** 4. Complete the function

```
decodeAExpr :: Integer → AExpr
```

which decodes arithmetic expressions. You may find `case` statements and the divMod function useful in writing short and readable code.

Solution

```
decodeAExpr :: Integer → AExpr
decodeAExpr n =
 case divMod n 5 of
  (q, 0) → EVar (decodeVar q)
  (q, 1) → EInt q
  (q, 2) → let (a1, a2) = unpair q in EBinOp AAdd (decodeAExpr a1) (decodeAExpr a2)
  (q, 3) → let (a1, a2) = unpair q in EBinOp ASub (decodeAExpr a1) (decodeAExpr a2)
  (q, 4) → let (a1, a2) = unpair q in EBinOp AMul (decodeAExpr a1) (decodeAExpr a2)
```

## Encoding Boolean expressions

*** 5. Follow the same strategy to define functions

```
encodeBExpr :: BExpr → Integer
decodeBExpr :: Integer → BExpr
```

that encode and decode Boolean expressions as natural numbers.

Solution

```
decodeBExpr :: Integer → BExpr
decodeBExpr n =
 case divMod n 5 of
  (q, 0) → BBool (decodeBool q)
  (q, 1) → BNot (decodeBExpr q)
  (q, 2) → let (b1, b2) = unpair q in BAnd (decodeBExpr b1) (decodeBExpr b2)
  (q, 3) → let (e1, e2) = unpair q in BComp AEq (decodeAExpr e1) (decodeAExpr e2)
  (q, 4) → let (e1, e2) = unpair q in BComp ALe (decodeAExpr e1) (decodeAExpr e2)
```

## Encoding While statements

** 6. Finally, use the very same strategy to define functions

```
encodeStmt :: Stmt → Integer
decodeStmt :: Integer → Stmt
```

that encode and decode While statements as natural numbers.

Solution

```
encodeStmt :: Stmt → Integer
encodeStmt SSkip = 0
encodeStmt (SAssign v e)  = 1 + 5 * pair (encodeVar v) (encodeAExpr e)
encodeStmt (SIte b s1 s2) =
 2 + 5 * pair (encodeBExpr b) (pair (encodeStmt s1) (encodeStmt s2))
encodeStmt (SWhile b s)   = 3 + 5 * pair (encodeBExpr b) (encodeStmt s)
encodeStmt (SSeq s1 s2)   = 4 + 5 * pair (encodeStmt s1) (encodeStmt s2)

decodeStmt :: Integer → Stmt
decodeStmt n =
 case divMod n 5 of
  (q, 0) → SSkip
  (q, 1) → let (v, e) = unpair q in SAssign (decodeVar v) (decodeAExpr e)
  (q, 2) →
   let (b, x) = unpair q
       (s1, s2) = unpair x
   in SIte (decodeBExpr b) (decodeStmt s1) (decodeStmt s2)
  (q, 3) → let (b, s) = unpair q in SWhile (decodeBExpr b) (decodeStmt s)
  (q, 4) → let (s1, s2) = unpair q in SSeq (decodeStmt s1) (decodeStmt s2)
```

## Test your code

I have provided three example ASTs for testing your code. The first one, namely

```
astExample1 :: Stmt
astExample1 =
  SSeq
    (SAssign "x1" (EInt 1))
    (SAssign "x2" (EBinOp AAdd (EInt 4) (EInt 2)))
```

is a manually-defined AST representing the While program `x1 := 1; x2 := 4 + 2`.

The second one uses the parser to parse the first two lines of the While code that swaps two integers:

```
astExample2 :: Stmt
astExample2 =
  fromRight SSkip $ parse $ unlines [
    "x1 := x1 + x2",
    "x1 := x2 - x1"
  ]
```

(If the parser fails for some unforeseen reason the AST for `Skip` is returned.)

Finally, the third example parses the full program that swaps two integers:

```
astExample3 :: Stmt
astExample3 =
  fromRight SSkip $ parse $ unlines [
    "x1 := x1 + x2",
    "x1 := x2 - x1",
    "x2 := x2 - x1"
  ]
```

** 7. Test your code using these three examples. What do you observe, and why?

Solution

The first two run smoothly, but the third one fails to run. As our code is fairly well-structured and has no obvious errors, the only possible culprit is the pairing function. Indeed, when we encode long statements into integers we produce very large numbers. Thus, when we compute an expression such as $\sqrt{2h}$, which is a floating-point number, we run the usual risk of *overflow*. The problem only appears when the statements get long enough, which in this case is shockingly short: only 3 instructions suffice!

**** 8. Propose a solution to the problem.

Solution

The solution is to use a more efficient pairing function—e.g. Regan's linear-time-constant-space pairing function—which does not involve floating-point calculations. However, that probably involves reading original research papers!