Artifact for the paper "Extracting total Amb programs from proofs"

Ulrich Berger, Hideki Tsuiki

GraySD.hs is a Haskell program explained in Section 7 of the paper. It makes executable the program `gtos` of Section 6 that translates infinite Gray code into signed digit representation by implementing the amb operator with the Haskell libraries `Control.Concurrent` and `Control.Exception`. The purpose of our implementation is to experimentally validate the program `gtos` (termination and correct result), and to demonstrate its nondeterministic behaviour.

This document annotates GraySD.hs for a better understanding of the program.

1 Nondeterminism

Using the primitives of the Haskell libraries `Concurrent` and `Exception` we can implement nondeterministic choice through a program `ambL` that picks from a list nondeterministically a terminating element (if exists). Although in our application we need only binary choice, we implement arbitrary finite choice since it is technically more convenient and permits more applications, e.g. Gaussian elimination (Section 8.4).

```
import Control.Concurrent
import Control.Exception

ambL :: [a] -> IO a
ambL xs =
  do { m <- newEmptyMVar ;
       acts <- sequence
                 [ forkIO (do { y <- evaluate x ; putMVar m y })
                    | x <- xs ] ;
       z <- takeMVar m ;
       x <- sequence_ (map killThread acts) ;
       seq x (return z)
     }
```

Comments:

- `newEmptyMVar` creates an empty mutable variable,

- `forkIO` creates a thread,
- `evaluate` evaluates its argument to head normal form,
- `putMVar m y` writes `y` into the mutable variable `m` provided `m` is empty,
- the line `seq x (return z)` makes sure that the threads are killed before the final result `z` is returned.

2 Extracting data

We define the domain D (Section 2) and a program `ed` on D ('extract data') that, using `ambL`, nondeterministically selects a terminating argument of the constructor Amb.

```
data D = Nil | Le D | Ri D | Pair(D, D) | Fun(D -> D) | Amb(D, D)

ed :: D -> IO D
ed (Le d) = do { d' <- ed d ; return (Le d') }
ed (Ri d) = do { d' <- ed d ; return (Ri d') }
ed (Pair d e) = do { d' <- ed d ; e' <- ed e ; return (Pair d' e') }
ed (Amb a b) = do { c <- ambL [a,b] ; ed c } ;
ed d = return d
```

`ed` can be seen as an implementation of the operational semantics in Section 3.

3 Gray code to Signed Digit Representation conversion

We read-off the programs extracted in the Sections 5 and 6 to obtain the desired conversion function. Note that this is nothing but a copy of the programs in those sections with type annotations for readability. The programs work without type annotation because Haskell infers their types. The Haskell types contain only one type D. Their types as CFP-programs are shown as comments in the code below.

From Section 5.

```
mapamb :: (D -> D)  -> D -> D  -- (B -> C) -> A(B) -> A(C)
         -- (A(B) is the type of Amb(a,b) where a,b are of type B)
mapamb = \f -> \c -> case c of {Amb(a,b) -> Amb(f $! a, f $! b)}

leftright :: D -> D   -- B + C -> B + C
leftright = \b ->  case b of {Le _ -> Le Nil; Ri _ -> Ri Nil}

conSD :: D -> D    -- 2 x 2 -> A(3)
```

```
         -- (2 = 1+1, etc. where 1 is the unit type)
conSD = \c -> case c of {Pair(a, b) ->
      Amb(Le $! (leftright a),
          Ri $! (case b of {Le _ -> bot; Ri _ -> Nil}))}
```

From Section 6.

```
gscomp :: D -> D  -- [2] -> A(3)
gscomp (Pair(a, Pair(b, p))) = conSD (Pair(a, b))

onedigit :: D -> D -> D   -- [2] -> 3 -> 3 x [2]
onedigit  (Pair(a, Pair (b, p))) c = case c of {
       Le d -> case d of {
              Le _ -> Pair(Le(Le Nil), Pair(b,p));
              Ri _  -> Pair(Le(Ri Nil), Pair(notD b,p))
        };
        Ri _  -> Pair(Ri Nil, Pair(a, nhD p))}

notD :: D -> D   -- 2 -> 2
notD a = case a of {Le _ -> Ri Nil; Ri _ -> Le Nil}

nhD :: D -> D  -- [2] -> [2]
nhD (Pair (a, p)) = Pair (notD a, p)

s :: D -> D    -- [2] -> A(3 x [2])
s p = mapamb (onedigit p) (gscomp p)

mon :: (D -> D) -> D -> D   -- (B -> C) -> A(3 x B) -> A(3 x C)
mon f p = mapamb (mond f) p
   where  mond f (Pair(a,t)) = Pair(a, f t)

gtos :: D -> D    -- [2]  -> [3]
gtos = (mon gtos) . s
```

4 Gray code generation with delayed digits

Recall that Gray code has the digits 1 and -1, modelled as `Ri Nil` and `Le Nil`. A digit may as well be undefined ($\perp$) in which case it is modelled by a non-terminating computation (such as `bot` below). To exhibit the nondeterminism in our programs we generate digits with different computation times. For example, `graydigitToD 5` denotes the digit 1 computed in 500000 steps, while `graydigitToD 0` does not terminate and therefore denotes $\perp$.

```
delay :: Integer -> D
delay n  | n > 1     = delay (n-1)
         | n == 1    = Ri Nil
```

```
         | n == 0    = bot
         | n == (-1) = Le Nil
         | n < (-1)  = delay (n+1)
bot = bot

graydigitToD :: Integer -> D
graydigitToD a | a == (-1) = Le Nil
               | a == 1    = Ri Nil
               | True      = delay (a*100000)
```

The function `grayToD` lifts this to Gray codes, that is, infinite sequences of partial Gray digits represented as elements of D:

```
-- list to Pairs
ltop :: [D] -> D
ltop = foldr (\x -> \y -> Pair(x,y)) Nil

grayToD :: [Integer] -> D
grayToD = ltop . (map graydigitToD)
```

For example, `grayToD (0:5:-3:[-1,-1..])` denotes the Gray code $\bot : 1 : -1 : -1, -1, \ldots$ where the first digit does not terminate, the second digit (1) takes 500000 steps to compute and the third digit (-1) takes 300000 steps. The remaining digits (all -1) take one step each.

5 Truncating the input and printing the result

The program `gtos` transforms Gray code into signed digit representation, so both, input and output are infinite. To observe the computation, we truncate the input to some finite approximation which `gtos` will map to some finite approximation of the output. This finite output is a nondeterministic element of D (i.e. it may contain the constructor `Amb`) from which we then can extract nondeterministically a deterministic data using the function `ed` which can be printed.

In the following we define the truncation and the printing of deterministic finite data.

Truncating $d \in D$ at depth n.

```
takeD :: Int -> D -> D
takeD n d | n > 0 =
  case d of
    {
      Nil        -> Nil ;
      Le a       -> Le (takeD (n-1) a) ;
      Ri a       -> Ri (takeD (n-1) a) ;
      Pair(a, b) -> Pair (takeD (n-1) a, takeD (n-1) b) ;
```

```
      Amb(a,b)   -> Amb(takeD (n-1) a, takeD (n-1) b) ;
      Fun _      -> error "takeD _ (Fun _)" ;
    }
            | otherwise = Nil
```

Showing a partial signed digit.

```
dtosd :: D -> String
dtosd (Le (Ri Nil)) = " 1"
dtosd (Le (Le Nil)) = "-1"
dtosd (Ri Nil)      = " 0"
dtosd _             = " bot"
```

Printing an element of D that represents a finite deterministic signed digit stream.

```
prints :: D -> IO ()
prints (Pair (d,e)) = putStr (dtosd d) >> prints e
prints Nil          = putStrLn ""
prints _ = error "prints: not a partial signed digit stream"
```

6 Experiments

As explained in Section 7, there are three Gray codes of 0:

$$\begin{aligned} a &= \bot : 1 : -1, -1, -1, \ldots \\ b &= 1 : 1 : -1, -1, -1, \ldots \\ c &= -1 : 1 : -1, -1, -1, \ldots \end{aligned}$$

and the set of signed digit representations of 0 is $A \cup B \cup C$ where

$$\begin{aligned} A &= \{0^\omega\} \\ B &= \{0^k : 1 : (-1)^\omega \mid k \geq 0\} \\ C &= \{0^k : (-1) : 1^\omega \mid k \geq 0\}. \end{aligned}$$

Our `gtos` program nondeterministically produces an element of A for input a, an element of $A \cup B$ for input b, and an element of $A \cup C$ for input c. As the following results show, the obtained value depends on the speed of computation of the individual Gray-digits.

Input b:

```
*GraySD> ed (takeD 50 (gtos (grayToD (1:1:[-1,-1..])))) >>= prints
 1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1 bot
```

Input c:

```
*GraySD> ed (takeD 50 (gtos (grayToD (-1:1:[-1,-1..])))) >>= prints
-1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 bot
```

Input a (demonstrating that the program can cope with an undefined digit):

```
*GraySD> ed (takeD 50 (gtos (grayToD (0:1:[-1,-1..])))) >>= prints
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 bot
```

Input b with delayed first digit:

```
*GraySD> ed (takeD 50 (gtos (grayToD (2:1:[-1,-1..])))) >>= prints
 0 0 1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1 bot
```

Same, but with more delayed first digit:

```
*GraySD> ed (takeD 50 (gtos (grayToD (10:1:[-1,-1..])))) >>= prints
 0 0 0 0 0 0 0 0 0 1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1 bot
```

Input $1, 1, 1, \ldots$ which is the Gray code of $2/3$:

```
*GraySD> ed (takeD 50 (gtos (grayToD ([1,1..])))) >>= prints
 1-1 1-1 1-1 1-1 1-1 1-1 1 0-1-1 1-1 1-1 1-1 1-1 1-1 bot
```

Same, but with delayed first digit:

```
*GraySD> ed (takeD 50 (gtos (grayToD (2:[1,1..])))) >>= prints
 0 1 1-1 1-1 1-1 1-1 0 1 1-1 1-1 1-1 1-1 1-1 1-1 bot
```

To see that the last two results are indeed approximations of signed digit representations of $2/3$, one observes that in the signed digit representation `0 1` means the same as `1-1` ($0 + 1/2 = 1 - 1/2$), so both results are equivalent to

```
 1-1 1-1 1-1 1-1 1-1 1-1 1-1 1-1 1-1 1-1 1-1 1-1 1-1 bot
```

which denotes $2/3$.

Note that since our experiments use the nondeterministic program `ed`, the results obtained with a different computer may differ from the ones included here. Our theoretical results ensure that, whatever the results are, they will be correct.