Adaptive Functional Implementation of MS Excel SpreadSheets

João Fernandes
jpaulo@di.uminho.pt

Departamento de Informática,University of Minho
Braga, Portugal

1 Introduction

An adaptive computation maintains the relationship between its input and output as the input changes. That is to say that output will be updated as response to input changes, while re-evaluation will only occur on program portions afected by the change. Adaptive programming is most useful in situations where input changes lead to relatively small changes in the output. In limiting cases one cannot avoid a complete re-computation of the output, but in many cases the results of the previous computation may be re-used to obtain the updated output more quickly than a complete re-evaluation. We are using Magnus Carlsson Adaptive Combinators library to model, in Haskell, a real spreadsheet adaptive system. This is, in fact, one of the classical incremental computation problems. Magnus Carlsson's Haskell library for incremental computing, by its hand, follows closely the implementation in the POPL 2002 paper "Adaptive Functional Programming", by Umut Acar, Guy Blelloch and Bob Harper. In this paper, Acar, Blelloch and Harper present an adaptivity mechanism based on the idea of a modifiable reference (or modifiable) and three operations for creating, reading and writing modifiables. A modifiable reference is essentially a write-once reference cell that records the value of an expression whose value may change as a (direct or indirect) result of changes to the inputs. Any expression whose value can change must store its value in a modifiable reference; such an expression is said to be changeable. Expressions that are not changeable are said to be stable; stable expressions are not associated with modifiables. Any expression that depends on the value of a changeable expression must express this dependence by explicitly reading the contents of the modifiable storing the value of that changeable expression. This establishes a data dependency between the expression reading that modifiable and the expression that determines the value of that modifiable. Magnus Carlsson's Haskell library provides a monadic interface, which doesn't need the write operation. In addition, the monadic types enforce correct usage, which means that a modifiable variable must be defined before it can be used. This is achieved within Haskell's type system plus some popular extensions, like multi-parameter classes and functional dependencies.

2 Implementation

When we thought about developing this project, we we're thinking about combining MS Excel's XML spreadsheet data saving with HaXml, and by that obtain a platform upon which we could build an Adaptive spreadsheet interpreter in Haskell. That is to say that we are using HaXml to convert a MS Excel (in fact, we are using Gnumeric, but the conversions from one's spreadsheets into the other's are simple) spreadsheet data specification and we are building a Haskell AST that represents the same specification. By then we obtain an abstract syntax tree of the following type:

```
data Gmr'Cells = Gmr'Cells Gmr'Cells_Attrs [Gmr'Cell]
```

where,

```
data Gmr'Cells_Attrs = Gmr'Cells_Attrs
{ gmr'CellsXmlns :: (Defaultable String)
}

data Gmr'Cell = Gmr'Cell Gmr'Cell_Attrs String

data Gmr'Cell_Attrs = Gmr'Cell_Attrs
{ gmr'CellXmlns       :: (Defaultable String)
, gmr'CellCol         :: String
, gmr'CellRow         :: String
, gmr'CellExprID      :: (Maybe String)
, gmr'CellValueType   :: (Maybe String)
, gmr'CellValueFormat :: (Maybe String)
, gmr'CellCols        :: (Maybe String)
, gmr'CellRows        :: (Maybe String)
, gmr'CellAst         :: (Maybe Formula)
}
```

This would, by its own, provide support for developing a non-incremental spreadsheet interpreter over ASTs of this type. As we are interested in obtaining an incremental one, we would have to work on data-structures where desired elements are placed in modifiables. This could be obtained by one of two aproaches: we could either change the parser to build ASTs of some other type, or use an auxiliary function to map ASTs over this data-type in ASTs over another data-type, with modifiable structure. By now, we are implementing the second aproach, and so we are mapping Gmr'Cell values into GnuExpr GnuCellRef values, where:

```
type InIO m a = m IO IORef a
type IOMod a  = InIO Modifiable a

data GnuExpr c  = Const (Int,Int) Float
                | Add (Int,Int) [GnuExpr c]
                | Mul (Int,Int) [GnuExpr c]
                | Sub (Int,Int) [GnuExpr c]
                | Div (Int,Int) [GnuExpr c]
                | Cell (Int,Int) c

data GnuCellRef = CR (Int,Int) (IOMod Float)
```

So, basically, what we are saying is that, for example, an expression may be a float constant, placed in a cell (represented by its matrix position). Yet, it is also possible to sum, multiply, divide and subtract multiple expressions. The diference between this representation and the non modifiable one is that now, when an expression is a reference to a cell, we are explicitly saying that its numeric value may change (obviously) when the refered cell is changed.

Having done this representation change, we must yet establish data dependencies between the spreadsheet cells. Namely, we must relate each cell that refers another one with the cell it refers, so that the adaptive mechanism is able to correctly propagate the changes made in referred cells. This process is done like the following: for each cell, we are creating a data-dependency between itself and the numeric value it holds (which is placed into a modifiable); this is achieved by doing:

```
newCell ::  (NewMod n IO IORef) =>
            GnuExpr GnuCellRef
            -> n IO IORef ((Int, Int),
                           (Modifiable IO IORef (GnuExpr GnuCellRef), IOMod Float))
newCell  expr = do
    c <- newMod (return expr)
    v <- newMod $ readMod c >>= eval
    let pos@(row,col) = getPos expr
    return (pos,(c,v))
```

, where the eval evaluation function is:

```
eval ::  GnuExpr GnuCellRef
      -> InIO Changeable Float
eval (Const _ x)        = return x
eval (Add   _ l)        = ap (return sum)      (sequence (map eval l))
eval (Mul   _ l)        = ap (return product)  (sequence (map eval l))
eval (Sub   _ l)        = ap (return subtract) (sequence (map eval l))
eval (Div   _ l)        = ap (return divide)   (sequence (map eval l))
eval (Cell  _ (CR _ f)) = readMod f
```

Mapping this transformation along the entire environment would result on any change produced in any spreadsheet cell being reflected on its value, without entire spreadsheet recalculation.

This newCell mapping performs slightly different for cell references, though.

When we have a cell reference, we are explicitly consulting the global environment in order to find whether the referred cell has already been computed. In the case that it has, we assing the (modifiable) value of the referred cell to the cell which refers it. This creates the needed data-dependency between a cell that refers another and this referred one.

So, suppose we have a cell **x** that refers the cell **y**. A data-dependency is created between **y** and its value, say **v**. Then, a data-dependency between **x** and **y** is established, by saying that the value of **x** is also **v**. Now, suppose we change **y** to **y'**. Then **v** will have to be re-evaluated, to **v'**, say. Since **x** depended on **v**, it will also change its value to **v'**.
In conclusion, a change made in a referred cell is correctly reflected on the cell that refers it.

So, instead of mapping the newCell operation over the environment, we are mapping the intoCell operation, which is trivial for all cases but cell references, and for this case it looks like:

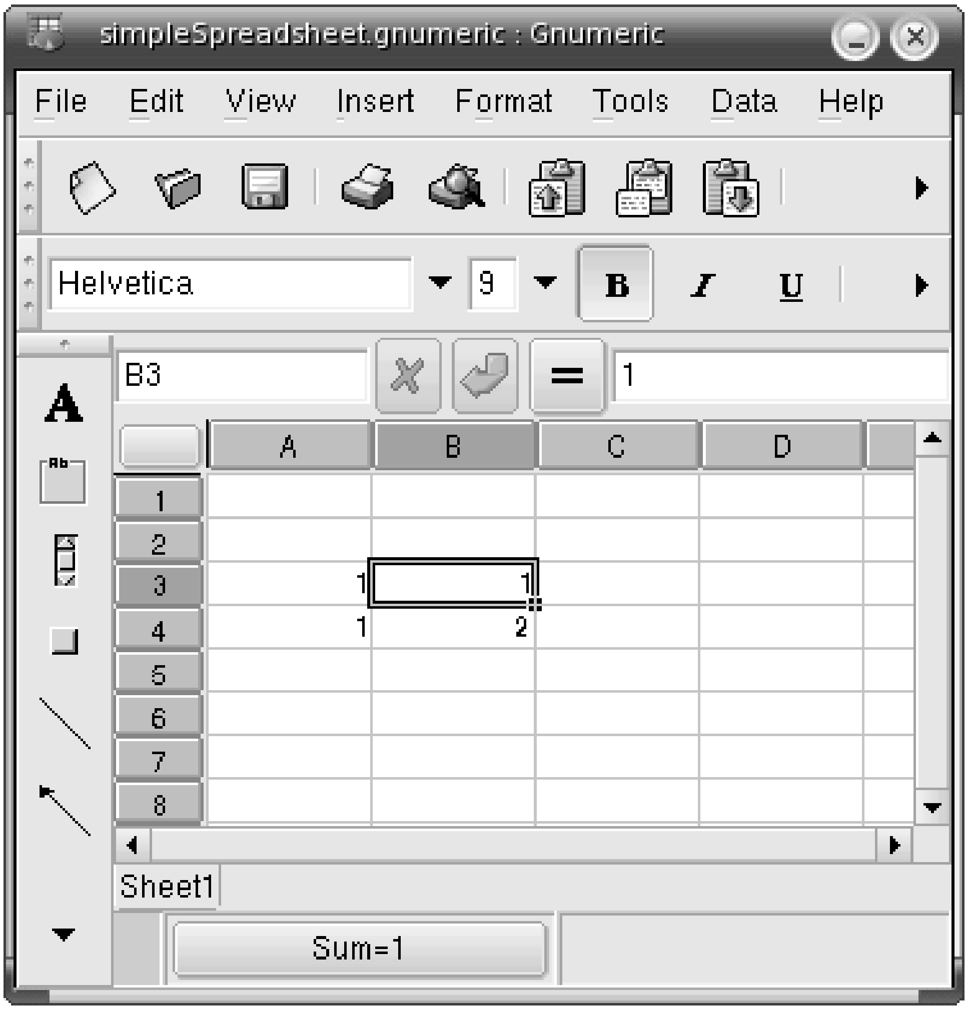
```
intoCell :: (InM m,Monad (m IO IORef),NewMod m IO IORef) =>
                [((Int,Int),(IOMod (GnuExpr GnuCellRef),IOMod Float))]
             -> [GnuExpr GnuCellRef]
             -> m IO IORef [((Int,Int),(IOMod (GnuExpr GnuCellRef), IOMod Float))]
intoCell l (c@(Cell pos (CR ref _)):cs) =
      case (lookup ref l) of Nothing -> intoCell l (cs++[c])
                             Just (expr, x)  -> do a <- newCell (Cell pos (CR ref x))
                                                   intoCell (a:l) cs
```

By this, the desired adaptive behaviour of the spreadsheet interpreter is achieved.

For simplicity in using the interpreter, we have also developed an interactive system for working in a spreadsheet-like environment within Haskell.

3 Running Example

We will now demostrate the functionality of the adaptive interpreter produced, by running a simple example. If we have the following spreadsheet:



The interactive interpreter would read it and display the following spreadsheet-like interactive environment:

```
1.0    Const (2,0) 1.0          1.0    Const (2,1) 1.0
1.0    Cell (3,0) (2,0)         2.0    Mul (3,1) [Const (3,1) 2.0,Cell (3,1) (2,1)]

Cell>
```

So, we can easily observe that the (2,0) and (2,1) cells hold the value **1**; the (3,0) cell points to the cell (2,0) and the (3,1) cell holds the multiplication of **2** and the content of the cell (2,1).

A change on the cell (2,0) should result on the change of itself and also on the change of the (3,0) cell, which refers it. Let's then try to do this, and change it to **5**, for example.

```
1.0     Const (2,0) 1.0             1.0     Const (2,1) 1.0
1.0     Cell (3,0) (2,0)            2.0     Mul (3,1) [Const (3,1) 2.0,Cell (3,1) (2,1)]

Cell> (2,0) Expr> 5
```

The result is the following:

```
5.0     Const (2,0) 5.0             1.0     Const (2,1) 1.0
5.0     Cell (3,0) (2,0)            2.0     Mul (3,1) [Const (3,1) 2.0,Cell (3,1) (2,1)]

Cell>
```

Using the same argument, any change on the (2,1) cell should be reflected on the (3,1). Let's now try this situation out.

```
5.0     Const (2,0) 5.0             2.0     Const (2,1) 2.0
5.0     Cell (3,0) (2,0)            4.0     Mul (3,1) [Const (3,1) 2.0,Cell (3,1) (2,1)]

Cell>
```

In both tested cases, the results obtained are the expected, in the sense that the adaptive mechanism updated values reflecting other values explicit changing. And this behaviour is obtained without the entire spreadsheet recalculation.

We also tried out an example where we create mutual dependent cell references. In a not obvious way, this situation is allowed, under our interpreter, because all 'existing' cells have a default value, which excludes the possibility of refering to an empty cell. But in fact, there's some strange behaviour of the adaptive mechanism going on.

Let's now consider the following example.

```
1.0     Const (2,0) 1.0             1.0     Const (2,1) 1.0
1.0     Cell (3,0) (2,0)            2.0     Mul (3,1) [Const (3,1) 2.0,Cell (3,1) (2,1)]

Cell>(2,1) Expr> c(3,1)
```

There's a reference in the cell (3,1) to the cell (2,1). We will try to create a reference to the cell (3,1) in the cell (2,1).

We get the following spreadsheet:

```
1.0     Const (2,0) 1.0             2.0     Cell (2,1) (3,1)
1.0     Cell (3,0) (2,0)            2.0     Mul (3,1) [Const (3,1) 2.0,Cell (3,1) (2,1)]

Cell>
```

If we now do some changes in other cells, like changing the value of the (2,0) cell to **2**, the spreadsheet becomes like:

```
2.0    Const (2,0) 2.0          4.0     Cell (2,1) (3,1)
2.0    Cell (3,0) (2,0)         2.0     Mul (3,1) [Const (3,1) 2.0,Cell (3,1) (2,1)]

Cell>
```

What seems to have happened is that, as it points to the cell (3,1),and the cell (3,1) holds **2*c(2,1)**, the cell (2,1) really could hold the value 4. But then the value of (2,1) would be **4**, so the value of (3,1) should be **8**, so this should also be the value of (2,1). And so on... Let's try another iteration of the change propagation that happens whenever a change is performed. For example, let's change the (2,0) cell to **3**. We get:

```
3.0    Const (2,0) 3.0          16.0    Cell (2,1) (3,1)
3.0    Cell (3,0) (2,0)         2.0     Mul (3,1) [Const (3,1) 2.0,Cell (3,1) (2,1)]

Cell>
```

What seems to be happening is that the **2*** operation is happening twice each time, one for each of the references. The wird thing is that the updating only happens in the last created reference.

4 Future Work

From the previously released version of this prototype tool, we have extended the adaptive mechanism in order to make it fully reliable. We have also improved our study of the adaptive library's functionality and usage.

Yet, we must keep studying it (and its aplication in our tool) in order to understand better the mutual dependent cell references.

The parser changes in order to deforest some unnecessary ASTs would also become quite handy.