Object-Oriented Functional Spreadsheets

Chris Clack and Lee Braine*

Department of Computer Science, University College London,
Gower Street, London WC1E 6BT, UK

Abstract. The spreadsheet is one of the most successful computer applications. This popularity derives from an intuitive user interface which both closely mimics traditional bookkeeping and allows non-programmers to develop simple numerical applications. Unfortunately, the current user interface is frustrating and limiting: we believe that the computational model can be *simplified* to improve usability for non-programmers, *extended* to provide additional functionality, *redesigned* to facilitate reuse (to improve performance and integrity), and *embedded* in an environment which supports a spreadsheet inheritance hierarchy. We propose a new spreadsheet paradigm which incorporates many functional programming features such as higher-order functions, a strong type system, curried partial applications, referential transparency and lazy evaluation. It also incorporates many object-oriented programming features such as a class hierarchy, inheritance, overloading, overriding, subsumption, and dynamic despatch on a distinguished object.

1 Introduction

The spreadsheet is one of the most successful computer applications. This popularity derives from the intuitive user interface which both closely mimics traditional bookkeeping and allows non-programmers to develop simple numerical programs. The user of a spreadsheet constructs a *spreadsheet application*, though it is not normally appreciated by the user that this is a form of applications programming. In fact, the success of the paradigm depends on the user's ability to construct applications *without* needing traditional programming skills.

Despite many advances in programming languages since the late 1970s, the spreadsheet paradigm has changed little in this time. Although some advances have been made (for example, working with multiple worksheets, defining name bindings, making spreadsheets available as components, linking cells to other components, and mouse-based range identification), the central computational model remains unaltered.

Unfortunately, the current user interface can be frustrating because there are many simple data-manipulation operations which can only be achieved through the application of programming skills (for example, constructing formulae which involve relative addressing, or writing spreadsheet macros). In particular, it is

* During the course of this work, Lee Braine was supported by an EPSRC research studentship and a CASE award from Andersen Consulting.

difficult to manipulate sequences or sets of ranges. Furthermore, the lack of simple mechanisms for re-use reduces performance (because formulae are typically copied many times), reduces integrity (because it is difficult to tell when a formula has been overwritten with a value), and hinders the development of spreadsheet applications (because it is difficult to develop new applications as extensions of existing applications).

From the perspective of the advanced spreadsheet user, application development is equally frustrating. Existing commercial spreadsheets such as Excel^{TM} and Lotus 1-2-3^{TM} are often criticized for the lack of proper abstraction mechanisms and for the shortcomings of their macro facilities[CR92, Lit90]. Furthermore, there is a limited type system and poor support for selection and iteration (including recursion).

We believe that many benefits can be derived from radically updating the spreadsheet computational model, whilst retaining the essence of the spreadsheet user interface. It might be argued that complex applications require programming skills and therefore a real programming language or a database system enriched with an advanced query language should be used. However, this view assumes that the limits of expressibility have already been reached as far as inexperienced users are concerned. We take the opposite view — we believe that the limits of expressibility have not yet been reached and that empowering inexperienced users is a worthwhile research aim. Indeed, the "first steps" outlined in this paper indicate how expressibility can be improved.

Since there is clearly a good correspondence between the computational model of the spreadsheet and that of a functional language (see Section 3), it is a natural step to attempt to extend spreadsheet functionality to encompass the features available from modern functional programming (FP): by incorporating FP features we aim to increase the simplicity, expressibility and power of spreadsheet programming. With an underlying FP model of computation we are also able to investigate new and more expressive user-interface operations such as the manipulation of visual spreadsheet regions as both l-values and r-values in formulae. This provides an alternative mechanism to relative addressing and avoids multiple copies of formulae.

We observe that it is difficult to organise, structure and group spreadsheets and therefore large-scale applications can be expensive and cumbersome to develop. Furthermore, spreadsheets are difficult to extend, modify and reuse, and large applications are correspondingly difficult to maintain. These issues could be ameliorated if spreadsheets supported the key concepts from modern object-oriented programming (OOP).

Both FP and OOP features can be utilised to provide a high degree of reuse, which improves performance and visual integrity; it also facilitates the constuction of audit-trails, which is often important for the financial sector.

Our approach involves viewing a spreadsheet system as a programming environment and using (a variant of) our new object-oriented functional language CLOVER [BC96, BC97a, BC97b, BC97c] to support the new computational model. A naïve approach might be merely to replace CLOVER's visual-

programming interface with a spreadsheet grid to support name bindings. However, spreadsheets must be "immediately reactive" in that all viewed cells have their current values displayed: as soon as one cell is modified then all dependent cells are re-evaluated — this is very different from the CLOVER edit-compile-run sequence. Thus, object-oriented functional spreadsheets present a significant challenge, not just in the combination of OOP and FP technology, but also in the design of the semantics of this new paradigm so that it retains the essential characteristics of spreadsheet programming.

In this paper we present the first steps in our design of a new spreadsheet paradigm. We discuss the problems that must be overcome in order to achieve this goal and present our design decisions for solving these problems.

2 Related Work

There is a large body of related work regarding the spreadsheet paradigm, though very little is directly relevant to the incorporation of both FP and OOP features. The closest pieces of related academic work are:

- FunSheet [DRV95] is a spreadsheet written in CLEAN. The expression language is also functional and higher-order. An interesting approach is taken in that columns are expressed as functions which are applied to a row index, which (together with higher order functions) provides a better mechanism for abstraction than that of relative addressing. However, FunSheet does not include any OOP features.
- Simple [Sta93] is an enhanced declarative spreadsheet which provides a constraint-based environment but does not address OOP issues;
- Davie and Hammond have investigated persistent hypersheets [DH96], but do not address OOP issues;
- the Generalised Spreadsheet Model of Yoder and Cohn [YC94, YC95, YC97] and the work of Wack [Wac95] are both discussed below.

Commercial products such as Excel^{TM} and Quattro Pro^{TM} provide support for viewing spreadsheets as components, but this does not change the computational model of the spreadsheet.

The integration of OOP and FP languages has received a lot of attention (see [BC96] for a survey of this related work), but most of these systems lose the key feature of referential transparency and none consider the application of the resulting technology to spreadsheets.

The use of declarative languages to implement spreadsheets has similarly received considerable attention [DW88, Wra86, HW94]. However, this is not directly relevant to our research since we are concerned with adding both FP and OOP features to the spreadsheet computational model (rather than merely using FP as an implementation vehicle).

Generalised Spreadsheet Model: Yoder and Cohn exploit implicit concurrency of spreadsheet operations. In [YC95], they extend the spreadsheet

paradigm allowing a block of cells to be: (i) associated with a function that takes parameters and returns a result, and (ii) recursively defined (i.e. a block may contain other blocks). However, their approach is different to ours in that they provide support for explicit inter-cell communication to facilitate concurrent evaluation of spreadsheet applications; furthermore, they allow blocks to access any value that is a child or grandchild of the parent block, which does not fit well with our framework. Their research is summarised in [YC97] with the definition of a Generalised Spreadsheet Model.

Wack: Like Yoder & Cohn, Wack [Wac95] is primarily interested in the implicit parallelism of spreadsheets. However, he adds several new features such as user-controlled dimensionality, infinite definitions, the separation of data-driven and demand-driven parts of the spreadsheet, and user-defined functions (by allowing cells to be λ-forms)

3 Design Issues

It has often been noted that the spreadsheet is a declarative system: a solution is specified using an equational style and there is no notion of control-flow. However, it is not immediately clear how modern FP features such as polymorphism, higher-order functions and lazy evaluation could be incorporated into a spreadsheet. Similarly, incorporating OOP features such as inheritance, classes and methods would seem to be problematic. In this short section we list some of the issues we have been forced to address at the design stage. The next section explains how we have attempted to resolve these issues.

- *Objects and Methods:* What is the correspondence between the worksheets and cells of a traditional spreadsheet and the objects and methods of OOP? Should a worksheet be an object, a method or neither? Is a cell an object, a method or neither?
- *Inheritance:* The unit of inheritance in OOP is the class, yet how can the notion of a class be incorporated into the traditional spreadsheet paradigm?
- *Type safety:* If there is inheritance, subtyping and dynamic despatch then how do we retain the referential transparency and type safety required by FP? how can these FP and OOP features be combined?
- *Parametric Polymorphism:* The inclusion of parametric polymorphism implies that parameters exist somewhere, but where? Where would such functions be defined? Would the user be able to define his or her own functions (methods)? Could these functions only be used in expressions inside a cell, or could cells or worksheets themselves in some way be parameterised?
- *Higher-order Functions:* Would higher-order functions only be available for use in the formulae within worksheet cells, or could worksheets themselves in some way be higher-order?
- *Lazy Evaluation:* How can lazy evaluation and potentially-infinite data structures be reconciled with "immediate reactivity"? The latter requires a form

of data-driven (strict) rather than lazy evaluation, so how would the value of an infinite data structure be displayed in a cell?
- *Relative Addressing:* Can FP provide a solution to the complexity of relative addressing modes? How can we provide the functionality of relative addressing without the disadvantages of copying and complexity? How can functionality and expressiveness be improved?

Perhaps the most daunting challenge is to find a simple unifying computational model that is intuitive to users of traditional spreadsheets. We therefore adopt an incremental approach; starting with a traditional spreadsheet system, we add new features one at a time and ensure that these do not affect the essential "look-and-feel" of the spreadsheet.

4 Evolving a New Spreadsheet Paradigm

4.1 Basic Technology

Syntax: We start by defining an abstract syntax for a simple idealized modern spreadsheet application (known as a "workbook" in Excel). A workbook consists of several worksheets; each worksheet consists of a grid of cells; each cell has a label and contains an expression. We define all function application to be prefix (to simplify the syntax) and assume that all `cellLabels` denote absolute addresses (leaving relative addressing to further work). In the remainder of this paper, we build on the following syntax definition as we add FP and OOP features:

```
project         :: workbook
workbook        :: bookName namedef* worksheet+
namedef         :: IDENTIFIER '=' expression
bookName        :: IDENTIFIER

worksheet       :: sheetName grid
sheetName       :: IDENTIFIER
grid            :: row+
row             :: cell+
cell            :: cellLabel expression
cellLabel       :: sheetName rowNum colName
rowNum          :: INTEGER
colName         :: IDENTIFIER

expression      :: application | cellLabel | LITERAL | UNDEFINED
application     :: OP arg*
arg             :: range | expression
range           :: cellLabel ':' cellLabel
```

Interactivity and referential transparency: As noted above, a spreadsheet specifies a collection of computations using an equational style (cells contain

either values or formulae) and there is no notion of control-flow. However, the system is highly interactive, with recalculation of values occuring every time a change is made to a cell. We find it helpful to view this sequence of modifications as a succession of static programs rather than as one continually-changing program — this interpretation allows us to view every part of the system (cells, worksheets and workbook) as being referentially transparent.

Evaluation: The interactive nature of spreadsheets results in a complex data-driven system in which any node in the data-flow graph can be updated either to modify the graph topology or to provide new data. The traditional data-driven evaluation semantics are that every time a cell (a node) is updated, that cell and all cells which depend on it (directly or indirectly) are re-evaluated. The evaluation of a cell's formula might cause it to access the value of another cell; in this case it uses the most recently calculated value and does not cause further (demand-driven) evaluation.

Problems often occur in traditional spreadsheets with cycles in the dependency graph. Our system is no different; a cyclic dependency could lead to an infinite loop. Just as with modern spreadsheets, we could detect some instances of loops (or at least detect situations that are likely to be caused by a loop) and issue some warning to the user.

4.2 Adding FP Features

In this subsection we progressively add several FP features to the basic spreadsheet: parameterised worksheets, a polymorphic type system, recursion, higher-order functions and curried partial applications. Lazy evaluation turns out to be the most difficult FP feature to add, and we leave discussion of lazy evaluation to further work at the end of the paper.

Parameterised worksheets: A workbook has been defined as a collection of worksheets; we now allow one or more of these worksheets to take parameters and to return a result. Thus, the spreadsheet application is split into two parts: (i) simple worksheets that do not take parameters (and which provide a traditional non-reusable grid), and (ii) worksheets that take one or more parameters and return a single result (which provide reusable "worksheet templates").

A parameterised worksheet is thus a function definition. This is similar to [YC95] where a block of cells may be associated with a function — however, our mechanism differs because the code for the function is given by the spreadsheet itself (whereas in [YC95] it is necessary to write separate code for the associated function). We believe that the grid format provides a useful structuring mechanism for the function body, wherein cells may contain expressions with references to the worksheet's parameters.

Each parameterised worksheet is given a name and the result of the worksheet is defined by the user to be an expression, which may be the value of one of the worksheet cells (this is similar to [YC95] where the first cell of a block is defined to

be the block's value). Note that (i) the expression may return a data-structure containing the values from many cells in the worksheet, and (ii) the value(s) returned are determined by the worksheet (callee) rather than by the caller of the worksheet, which improves encapsulation.

The changes required to the spreadsheet syntax are as follows (the rest of the syntax remains unchanged):

```
simpleworksheet :: sheetName grid
worksheet       :: sheetName parameter+ result grid
parameter       :: IDENTIFIER
result          :: expression
application     :: OP arg* | sheetName arg+
```

A polymorphic type system: Traditional spreadsheets are restricted to a few base types such as number, string, bool and location (i.e. a cell label). We extend this to include the base types int, real, nat, string, bool, and new aggregate types of tuple and list. Note that cell location identifiers (such as "B3") which appear in formulae have the type of the cell to which they refer.

Each type supports a collection of primitive operations (including a `show` operator that is silently applied to the result of every cell). At this stage we assume that lists are finite.

Parameterised worksheets have function type and we extend the type system to include parametric polymorphism. We provide polymorphism through the use of a type variable which can range over all types. Abstract data types are left to a later section in which we develop an object-oriented type system that identifies classes with types and allows user-defined classes.

We define a type syntax and enforce type declaration for worksheets and cells as follows:

```
type            :: 'int' | 'real' | 'nat' | 'string' | 'bool'
                 | 'tuple' type+ | 'list' type
                 | typeVariable | type '->' type
typeVariable    :: TYPE_IDENTIFIER

worksheet       :: sheetName parameter+ result type grid
cell            :: cellLabel type expression
expression      :: application | cellLabel | LITERAL
                 | '(' expression item* ')'
                 | '[]' | '[' expression item* ']'
                 | UNDEFINED
item            :: ',' expression
```

Recursion: Worksheets are allowed to make recursive reference to themselves. However, the data-driven evaluation semantics of a spreadsheet require the safeguard that recursive definitions always terminate.

We choose to restrict our system (through syntactic constraints) to primitive recursion that is guaranteed to terminate. This restricts the computational

power of the system, yet in the next section we add higher-order functions and this combination of primitive recursion plus higher-order functions provides a system of adequate computational power (see [Tur95]). The syntactic constraint we apply is that *each recursive function must have a constructed type as its parameter of recursion, the function must not recurse when the base case of the constructed type is detected, and each recursive call must be on a syntactic subcomponent of its parameter of recursion.* In this, we follow the lead of [Tur95].

Higher-order functions: In order to ensure that primitive recursion does not restrict us to a small set of computable programs, we extend our spreadsheet system so that it is legal to pass values of function type (i.e. names of parameterised worksheets or built-in operators, and partial applications as defined in the next section). Similarly, values of function type can be stored in cells and returned as the result of a worksheet. These are incorporated by supporting bracketed function types and expressions which can be worksheet names:

```
type            :: ... | '(' type '->' type ')'

expression      :: ... | sheetName
```

Curried partial applications: We have already defined our syntax to allow worksheets and built-in operations to be applied to multiple arguments in Curried form: we now extend the semantics to allow partial application.

A cell may now produce a partial application (of function type) and may contain a value that is the name of a parameterised worksheet or of a built-in operation, but note that we do *not* introduce generalised function abstraction and so cells cannot be λ-forms (see [BC96] for explanation).

We modify the syntax to support curried partial application by allowing any expression to be applied in function position (all the other required changes are in the semantics, which we do not present in this paper).

4.3 Adding OOP Functionality

We add OOP functionality to the above system in a similar way (and using similar design decisions) as we added OOP to FP to produce the CLOVER language [BC96]. We will not repeat the arguments of [BC96] here, but merely observe that it is possible to reconcile FP with the key aspects of OOP except for the issue of object identity, which in CLOVER must have copy-semantics rather than having mutable internal state.

For our object-oriented functional spreadsheet system we extend the user interface with a Smalltalk-like environment with browsers for the workbook hierarchy, attribute declarations and worksheet declarations, plus a worksheet editor providing a standard spreadsheet "look-and-feel" (see Figure 1).
The new syntax for our object-oriented functional spreadsheet system is presented below:

```
project         :: simpleworksheet* namedef* workbook
simpleworksheet :: sheetName grid
namedef         :: IDENTIFIER '=' expression

workbook        :: bookDefn workbook*
bookDefn        :: bookName attribute* worksheet+
attribute       :: IDENTIFIER type
bookName        :: IDENTIFIER

worksheet       :: sheetName parameter+ result type grid
sheetName       :: IDENTIFIER
parameter       :: IDENTIFIER
result          :: expression
grid            :: row+
row             :: cell+

cell            :: cellLabel type expression
cellLabel       :: sheetName rowNum colName
rowNum          :: INTEGER
colName         :: IDENTIFIER
expression      :: application | cellLabel | sheetName | UNDEFINED

application     :: expression arg+
arg             :: range | expression
range           :: cellLabel ':' cellLabel

type            :: typeVariable
                 | type '->' type | '(' type '->' type ')'
                 | bookName type*
typeVariable    :: TYPE_IDENTIFIER
```

We define a class to be a collection of parameterised worksheets which share some common attributes. An object is th reification of a class: that is, an instance of a class where the attributes have been given concrete values. This approach contrasts with [YC95] where a single block of cells (i.e. a single worksheet) is identified as an object.

A parameterised worksheet now corresponds to a method and what was previously a `namedef` is now an attribute with a defined type and scope that is local to the class (now called the "workbook"). All parameterised worksheets must take at least one argument which is the distinguished object (DO). When a worksheet name is applied, the DO determines dynamically in which workbook the actual worksheet can be found (the same worksheet name can be used in many different workbooks). Within a worksheet, arguments may be referred to as `arg1`, `arg2` etc., with the the last argument (which is defined to be the DO) always referred to as `self`.

Unparameterised worksheets do not fit well within a class structure (they do not have a DO) and so they are lifted to the top level — to use OOP terminology, they may be thought of as a collection of "invocations".

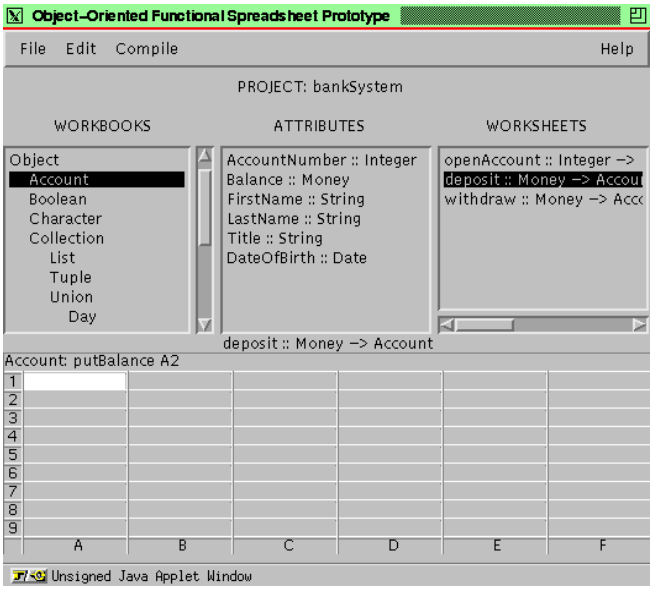

Fig. 1. Prototype spreadsheet environment

We also provide top-level bindings with global scope. Since a workbook is a class, it defines a new type and our type syntax is extended to allow reference to workbook names as types. To support container classes (e.g. a tree of int, list of char), we allow workbook names to take zero or more type parameters.

We define a class hierarchy with inheritance to structure spreadsheet applications, organise workbooks and promote reuse. This is achieved syntactically by defining a project to be a hierarchy of workbooks, each with its own definition and zero or more sub-workbooks. A newly-defined class can override the definitions of inherited worksheets (as long as the type does not change) and can provide multiple overloadedings for newly-defined worksheets (as long as each is uniquely distinguishable according to its type). The class hierarchy is initialised with definitions of classes for each of the built-in types: this simplifies our type syntax since every type is either a class or a function type. An object is the reification of a class; values are given to the class attributes upon instantiation.

Parameterised worksheets are dynamically despatched according to their DO. The class hierarchy also provides a subtype relation, with subsumption occuring during the application of a parameterised worksheet to its arguments.

Objects are instantiated in worksheet cells by applying the class constructor (there must be at least one for each class) to the arguments. Every class must have a `show` method which produces a displayable representation of its internal state. The default `show` method (defined in the class at the root of the class hierarchy) returns a default message: the `show` method for an object of class `int`

returns the value of the integer which is stored in an internal attribute.

5 Relative Addressing and Regions

Traditional spreadsheets use both absolute and relative addressing inside cell formulae to reference the values held in other cells. Relative addresses are interpreted as (row and column) offsets from the current cell; the assumption is that a single formula can be copied into one or more other cells and the relative addresses will in each case be offsets from the cell that contains a copy of the formula.

We hold that the use of relative addressing and the copying of formulae is undesirable because (i) it does not support abstraction (attention remains focussed at the level of the cell rather than at the structure of the numerical model), (ii) it does not support reuse (if a change must be made to the formula then new copies must be created all over again), (iii) it is extremely difficult to detect whether one of the formulae has been overwritten with a basic value, and (i) it is an unnecessary waste of memory to store the many copies.

Furthermore, [Hen95] notes that there are many desirable operations on spreadsheets that are easy to specify but difficult to program (and are beyond the grasp of the inexperienced user). These tend to be high-level operations that work on the structure of the data; there are no compensatory features to facilitate such high-level operations and so much of the functionality of spreadsheets is denied to the inexperienced user (and made difficult for the experienced user). Thus, one of the most important aims of our work on updating the computational model of the spreadsheet is to replace the concept of relative addressing and formula copying with something more suitable.

We are currently investigating a new syntax for expressing *regions* of cells. We believe that the use of higher-order functions over regions can reduce much of the complexity inherent in current relative addressing. We start by defining regions in a very similar way to traditional spreadsheet ranges - that is, two cell addresses separated by a colon. If the two addresses are on the same row (or column), this is interpreted as a list of cells (one-dimensional): if they are on different rows and columns then this is interpreted as a (row-major) list of lists of cells (two-dimensional):

`[A1:A5]` *is a 1-D vertical list of five cells*
`[A1:G1]` *is a 1-D horizontal list of seven cells*
`[A1:C3]` *is a 2-D list of lists containing nine cells*

What makes regions different from ranges is that regions can be used as *l*-values as well as *r*-values. To make a specific comparison, in many current spreadsheets it is possible to define a formula in one cell and then arrange for all of the cells in a range to receive a separate copy of the formula (with relative addresses mapped appropriately): by contrast, in our system all of the cells in a region can have their values defined by a single function. In the former, each cell is a

separate l-value for each separate copy of the function, whereas in the latter the entire region is the l-value for a single function.

This requires no extra modification to the user interface for a single cell; indeed, operations on a single cell appear unchanged. However, a new mechanism is added so that the user can bind a formula to a region of cells. Note that the type and "shape" of the right-hand-side of the definition must match the type and "shape" of the left-hand-side. Thus, defining a formula for the region [A1:A5] will require an expression on the right-hand-side that returns a list of five values (which will then be mapped one-to-one onto the cell locations contained in the region on the left-hand-side). This simple measure of using regions as l-values eliminates formula copying and eliminates the need for relative addressing. [2]

By providing higher-order functions which work on lists, it is now possible to express succinctly relationships of medium complexity:

```
[A1:A5] = map (*2) [B1:B5]
[A1:G1] = repeat 7 (sum (map (*3) [A1:A9]))
```

The real power with regions is achieved by providing a compact syntax for sequences of regions, in a similar way to Miranda's [3] list-comprehensions. We propose a new feature called *region-comprehensions* which draw their motivation from [Hen95] and which also bear some relationship to Scholz's "WITH-loops" [SCH97] (which provide a variant of ZF-expressions over arrays). This feature is at a very early stage of development, so for the purposes of this paper we merely provide the following examples (which may be used either as l-values or r-values):

```
[ [A1:D1], [E1:H1] .. [Q1:T1] ]
```
is a collection of 1-D regions all on the same row

```
[ [A1:D1], [C1:F1] .. [I1:L1] ]
```
is a collection of overlapping 1-D regions all on the same row

```
[ [A2:C4], [B3:D5] .. [D5:F7] ]
```
is a collection of overlapping 2-D regions

In the above expressions a sequence of regions is defined by example. There are two base regions and a terminating region. Given two base regions $[a_i m_i{:}a_j m_j]$ and $[b_i n_i{:}b_j n_j]$ four offsets are calculated as $(o^1, o^2, o^3, o^4) = (b_i - a_i, n_i - m_i, b_j - a_j, n_j - m_j)$.

The region comprehension is interpreted as a sequence of regions where the next region in the sequence is determined by applying the offsets to the previous region, so that for a previous region $[An : Bm]$ the next region in the sequence would be $[(A + o^1)(n + o^2) : (B + o^3)(m + o^4)]$. The sequence terminates when the final region is encountered.

---

[2] Of course, cell addresses (both l-values and r-values) will have to be recomputed when a row or column is added or deleted.

[3] Miranda is a trademark of Research Software Limited

Clearly, the above region comprehensions are very simple. We are currently investigating the extension of region comprehensions to incorporate generators, filters and recurrence relations as found in list comprehensions. The further development of regions and region-comprehensions is a key area for future work.

5.1 Examples

The following two examples demonstrate the use of a higher-order function together with region comprehensions.

Quarterly and Annual Profits:

```
    A   B   C   D   E   F   G   H   I   J   K   L   M   N   O   P
  ---------------------------------------------------------------
1 | 1993            1994            1995            1996
2 | Q1  Q2  Q3  Q4  Q1  Q2  Q3  Q4  Q1  Q2  Q3  Q4  Q1  Q2  Q3  Q4
3 |  3  12  12   4   7  15  28  12  15  39  30  15  10  10  10  10
4 |
5 |'93 '94 '95 '96
6 | 31  62  99  40
```

The data in row 3 of the above worksheet represents quarterly profit figures for a company. This row can be viewed as a sequence of non-overlapping 1-D regions `[A3:D3]`, `[E3:H3]` etc. The data in row 6 represents the annual profit figures: traditionally, this would be achieved by constructing a formula in cell A6 and then copying that formula to the cells B6, C6 and D6 using a mouse select and a paste command. However, we note the following:

1. The formula is not simple. For example, an obvious mistake is to use the formula `sum(A3:D3)` and copy this to the other calles. However, the relative addressing translation would cause B6 to be set to the formula `sum(B3:E3)` (which is incorrect). The required formula (here, in Excel) requires an appreciation of both absolute and relative addressing which is probably beyond the capability of inexperienced users:
   ```
   SUM(OFFSET($A$3,0,4*(COLUMN(A3)-1),1,4))
   ```
2. If a change must be made to the formula, the copy-select-paste procedure must be repeated;
3. It is very difficult to detect whether one of the cells in row 6 has been inadvertently or deliberately replaced with a value instead of a copy of the formula.

By contrast, in our example row 6 has been derived through the use of a single formula with no copying. Furthermore, our formula is simpler (noting that each region is selected with the mouse):

[A6:D6] = map sum [[A3:D3], [E3:H3] .. [M3:P3]]

Changing the spreadsheet so that the annual sums are held in a vertical (rather than horizontal) region of cells is also easier with region comprehensions:

```
[A6:A9] = map sum [ [A3:D3], [E3:H3] .. [M3:P3] ]
```

versus

```
SUM(OFFSET($A$3,0,4*(ROW(A6)-ROW($A$6)),1,4))
```

Two-Dimensional Moving Sum: The numbers in the following spreadsheet provide the underlying data which must be analysed. The required analysis is to generate a two-dimensional moving sum across the leading diagonal. That is, we require the sum of the nine values in the region [A1:C3] followed by the sum of the nine values in the region [B2:D4] and so on. Note that these regions are overlapping. The sums are presented in the final row: this is again difficult to achieve in traditional spreadsheets (relative addressing would make it easy to print the sums in a diagonal line of cells starting at A10 and ending at F15, but to print the sums all on one row requires programming skill). Here is the Excel solution:

```
SUM(OFFSET($A$1,COLUMN(A1)-1,COLUMN(A1)-1,3,3))
```

If we allow the built-in primitive `sum` to be overloaded so that it sums a list of lists of numbers as well as a list of numbers, then our solution is much simpler:

```
[A10:F10] = map sum [ [A1:C3], [B2:D4] .. [F6:H8] ]
```

	A	B	C	D	E	F	G	H
1	3	4	7	13	10	3	1	23
2	7	13	10	3	1	23	89	24
3	32	7	19	13	10	3	1	28
4	1	3	7	23	34	2	9	91
5	5	2	5	93	37	8	8	43
6	9	9	5	36	43	1	3	93
7	2	9	1	85	56	9	2	21
8	2	3	9	47	34	2	9	53
9								
10	102	98	241	277	167	193		

6 Project Status and Further Work

The design presented in this paper has resulted from our work in the area of object-oriented functional programming (OOFP). The notion of an object-oriented functional spreadsheet is both interesting in its own right and is used as a *driving application* for the development of our OOFP language, CLOVER [BC96]. We are currently implementing a prototype of this design and are collaborating with the University of St.Andrews to take these ideas further. We identify three areas in particular for further work:

Semantics: At present we have an informal semantics and our most pressing "next step" is to define formally the semantics of our new computational model.

Regions and Region Comprehensions: Our region comprehensions are currently very simple, but they provide a substantial improvement over relative addressing and formula-copying. We need to express formally the semantics of both regions and region-comprehensions, as well as investigating non-contiguous regions, more powerful comprehensions, and other operations on regions.

Demand-driven evaluation semantics: The inclusion of lazy evaluation into the spreadsheet system is rather problematic because of the unusual evaluation and display requirements (every cell's value is visible and may be recalculated as soon as any change is made). Although the ability to manipulate potentially-infinite data will bring improved expressiveness, such values cannot be displayed and the data-driven evaluation would seem to destroy any hope for laziness.

7 Summary and Conclusion

The central computational model of the spreadsheet paradigm has changed little during the past 20 years. We have identified some of the benefits that could be derived from incorporating modern FP and OOP features into this model. We discussed some key issues of such an integration and then presented our initial steps towards a novel spreadsheet design with the addition of FP and OOP features, together with the use of regions and region comprehensions to obviate the need for relative addressing.

References

[BC96] L. Braine and C. Clack. Introducing CLOVER: an Object-Oriented Functional Language. In W. Kluge, editor, *Implementation of Functional Languages, 8th International Workshop (IFL'96), Selected Papers*, LNCS 1268, 1–20, Springer-Verlag, 1996.

[BC97a] L. Braine and C. Clack. An Object-Oriented Functional Approach to Information Systems Engineering. In *Proceedings CAiSE'97 4th Doctoral Consortium on Advanced Information Systems Engineering*, 1997.

[BC97b] L. Braine and C. Clack. Object-Flow. In *Proceedings 13th IEEE Symposium on Visual Languages (VL'97)*, 418–419, 1997.

[BC97c] L. Braine and C. Clack. The CLOVER Rewrite Rules: A Translation from OOFP to FP. In *Proceedings 9th International Workshop on Implementation of Functional Languages (IFL'97)*, 467–488, 1997.

[CR92] Casimir and Rommert. Real Programmers Don't Use Spreadsheets. *ACM SIGPLAN Notices* 27(6), 10–16, 1992.

[DH96] A. Davie and K. Hammond. Functional Hypersheets. In *Proceedings 8th International Workshop on Implementation of Functional Languages (IFL'96)*, 39–48, 1996.

[DRV95] W. De Hoon, L. Rutten, and M. Van Eekelen. Implementing a Functional Spreadsheet in Clean. *Journal of Functional Programming*, 5(3):383–414, 1995.

[DW88] W. Du and W. Wadge. An intensional language as the basis of a 3D spreadsheet design. In *Proceedings IEEE International Conference on Computer Languages 1988 (ICCL'88)*, 2–9, 1988.

[HW94] B.Harvey and M.Wright. Simply Scheme: Introducing Computer Science. *MIT Press*, 1994.

[Hen95] D. Hendry. Display-Based Problems in Spreadsheets: A Critical Incident and a Design Remedy. In *Proceedings 11th International IEEE Symposium on Visual Languages (VL'95)*, 284–290, 1995.

[Lit90] C. Litecky. Spreadsheet Macro Programming: a Critique with Emphasis on Lotus 1-2-3. *Journal of Systems and Software*, 13(3), 197–200, 1990.

[SCH97] S. Scholz. With-loop Folding in Sac-Condensing Consecutive Array Operations. In *Proceedings Implementation of Functional Languages Workshop IFL'97*, 225–242, 1997.

[Sta93] M. Stadelmann. A Spreadsheet based on Constraints. In *Proceedings 6th Symposium on User Interface Software and Technology (UIST'93)*, 217–224, 1993.

[Tur95] D. Turner. Elementary Strong Functional Programming. In P. Hartel and M. Plasmeijer, editors, *Proceedings First International Symposium on Functional Programming Languages in Education (FPLE'95)*, LNCS 1022, 1–13, Springer-Verlag, 1995.

[Wac95] A. Wack. Partitioning Dependency Graphs for Concurrent Execution: A Parallel Spreadsheet on a Realistically Modelled Message Passing Environment. *PhD Thesis*, Department of Computer and Information Sciences, University of Delaware, 1995.

[Wra86] S. Wray. Implementation and Programming Techniques for Functional Languages. *Phd Thesis*, University of Cambridge, 1986.

[YC94] A. Yoder and D. Cohn. Real Spreadsheets for Real Programmers. In *Proceedings IEEE International Conference on Computer Languages 1994 (ICCL'94)*, 1994.

[YC95] A. Yoder and D. Cohn. A Framework for Complete Spreadsheet Languages. Technical Report, Distributed Computing Research Lab, University of Notre Dame, Indiana, USA, 1995.

[YC97] A. Yoder and D. Cohn. Domain-Specific and General-Purpose Aspects of Spreadsheet Languages. In *Proceedings Workshop on Domain-Specific Languages*, 1997.

This article was processed using the LaTeX macro package with LLNCS style