Haxcel: A Spreadsheet Interface to Haskell

Björn Lisper and Johan Malmström

Dept. of Computer Engineering, Mälardalen University, P.O. Box 883, SE-721 23 Västerås, SWEDEN
bjorn.lisper@mdh.se, jom@mail.jcd.se

Abstract. The spreadsheet paradigm offers a fast interactive loop, where the effects of updates to definitions and data are immediately visible. This makes the paradigm attractive for program development, where redefinitions can be immediately tested and the results displayed. We have designed a simple, compiler-independent spreadsheet interface to Haskell, where cells host Haskell definitions. Spreadsheets are also used for high-level array calculations. In order to meet that demand we have designed and implemented an extended array library for Haskell, which provides a number of typical array-language facilities. Together, the interface and the array library provide an interactive environment that can be used both for development of general Haskell code and for array-oriented spreadsheet calculations.

1 Introduction

Spreadsheet systems are very popular. They are used for small calculations, but also for serious application programming in areas like science and administration, when there is a need to perform calculations where fast and visually comprehensible feedback is more important than computational speed. A big reason for the popularity is the very tight definition-eval-display loop, where the effects of a redefinition of an entity are immediately seen. This makes the spreadsheet paradigm attractive, compared with traditional compiled languages where the compile-eval-print loop is considerably more tedious and where results are harder to present in a way that a human can grasp quickly.

Spreadsheets support an array-oriented style of computing. Sections of spreadsheets can be selected, and collective operations like summation can be applied to these. This makes it convenient to express many calculations. Without doubt, part of the popularity of spreadsheets can be attributed to this. The spreadsheet languages are also typically without side-effects. This provides a simple and intuitive evaluation model.

However, the common spreadsheet systems have weaknesses. The languages are typically very restricted: for instance, recursion is invariably not allowed, and only a few basic datatypes are usually supported. Some other common weaknesses are discussed in Section 1.

One possible remedy is to put a spreatsheet interface on top of a full-fledged programming language. Pure functional languages are prime candidates, since they are side-effect free. We have made such an interface, tentatively named “Haxcel”, for Haskell. The idea to have functional languages as formula languages in spreadsheets is not new [7, 8, 10], but to our knowledge this has not been done for Haskell before (although a Haskell-based interface has been outlined in [7]). We think Haskell is an interesting candidate for this, both as a strong language in general, and since it is relatively easy to define domain-specific sublanguages within it. We define an extended array library, which provides a little array language within Haskell. Thus, Haxcel can be used as a spreadsheet system with a more powerful language, but it can also be seen as a convenient environment for Haskell programming. (Note that the extended array library and the interface are independent: the library can be used without the interface, and vice versa.)

Haxcel is a standalone component that uses `hmake` to invoke the Haskell compiler at hand. It can thus be seen as a more visual relative to purely text-based interfaces such as hi [24].

The rest of this paper is organized as follows. In Section 2, we review the common spreadsheet model and identify some of its shortcomings. Section 3 describes the current interface and the design objectives for it. In Section 4 we describe the implementation of the interface. Section 5 describes the extended array library. Section 6 contains a spreadsheet programming example. Section 7 presents some experimental results. In Section 8 we give an account for related work, and Section 9 wraps up the paper with conclusions and ideas for further investigation. A more in-depth description of Haxcel and its implementation will be found in [18].

2 The Common Spreadsheet Model, And Some of Its Problems

In the typical spreadsheet model, a 2-D array of cells is used to visualize a set of interrelated values. Each cell has a 2-D coordinate and a definition of its value, possibly in terms of other values (referred to by their coordinates). A formula language is used to define values. A value can also be undefined: the spreadsheet must thus give some meaning to definitions involving undefined values. A definition or re-definition of a value results in a prompt recalculation of all dependent values.

Apparently, this mix of textual definition and visual feedback is very attractive for a range of tasks where calculations are done over and over again, on varying sets of data. Small changes in input data can be rapidly simulated by altering some definitions: the effects are immediately seen. Thus, the spreadsheet model has become very popular in management and economics, where simple economical models can be easily simulated under different assumptions. The two-dimensional layout encourages the organization of data into tables, which is beneficial for the regular processing of larger data sets. Therefore, spreadsheets are also popular for tasks like processing of measured data in laboratories.

In the spreadsheet model described above, definitions are given to coordinates of cells (e.g., “`A2`” in an Excel spreadsheet). This has some problems. A small problem is that the names of these coordinates often are poor mnemonics. A bigger problem is that the “tabular” and “definitional” uses of coordinates sometimes clash. “Tabular” use of spreadsheets can, for instance, require that rows or column are inserted in order to update a table that is embedded somewhere in the spreadsheet area. This will offset a number of definitions, so they now are associated with new coordinates. Even though most spreadsheets can update references to coordinates in definitions automatically, to compensate for this, it provides a burden for the programmer to keep track of the changing names of defined entities. Worse still is that insertion or deletion of rows or columns, in order to update one embedded table, can interfere with other tables or definitions that just happen to share the same rows or columns, see Fig. 1. In such a situation, a very tedious re-editing of the spreadsheet can be necessary.

	A	B	C	D	E	F	G
1		Salary	Other $				
2	Alan	17000	9			speed	strength
3	Bob	44000	99		THX99	47	95
4	Dave	23000	999		AGP	512	173
5					MPH	70	0

	A	B	C	D	E	F	G
1		Salary	Other $				
2	Alan	17000	9			speed	strength
3	Bob	44000	99		THX99	47	95
4	Charlie	19000	9999				
5	Dave	23000	999		AGP	512	173
6					MPH	70	0

Fig. 1. Interference between tables when inserting a row for “Charlie” in the first table.

Another, related problem, observed also in [6, 8], occurs when spreadsheet definitions are copied between cells. Very often, the copying is done to implement a similar computation on, say, a different row of a table than the original one. A convenient treatment of coordinates in the definition is then to let them “move”, so they have the same positions relative to the new coordinate of the copied definition. Therefore, spreadsheets tend to have this behaviour as default. But sometimes one would like to keep coordinates “absolute”, so they don’t move but refer to the same entity before and after copying. Spreadsheet systems like Excel provide a syntax that lets the programmer override the default behaviour, but the ambiguity of “copying semantics” is still a source of many bugs in spreadsheet programming.

A solution to the problems mentioned above is to create a clear separation between the use of spreadsheets to define entities and promptly visualize the results, and the use of spreadsheets for tabular calculations. Cells can be given general names rather than just coordinates, and one can have a separate cell for each name binding. Arrays can be used for tabular data. A change to the index range for one array, for instance to add/delete a row or column, will then not affect other arrays. If the formula language is sufficiently rich in array computing primitives, for instance to specify a similar computation being repeated over a range of array sections, then there will be less need to copy and paste definitions. Haxcel was designed to provide a solution of this kind.

3 The Spreadsheet Interface

Our spreadsheet interface Haxcel makes it possible to create or update Haskell declarations, and immediately view the results. It operates against a Haskell module at a time. Once a module is opened, all the declarations in the module are available through a list with one entry per declared name. Existing declarations can be edited or removed, and new definitions are created through a pulldown menu that offers a choice of the different kinds of Haskell declarations. Each declaration has an *editing window*, where it can be changed through a simple text editor. A value binding can also have a *value window*, which displays its current value. These windows can be freely opened, closed, and positioned. The evaluation of all open value windows, with the current declarations, is triggered by a button. See Fig. 3, which shows a possible GUI state for the module in Fig. 2. The GUI state of the module, including positioning information and information about whether windows are open or not, is saved together with the module contents. Thus, the user can persistently arrange the display of declarations and values for maximal clarity, similarly to how entities can be positioned in the grid of a conventional spreadsheet. The user can select to keep open only those windows that matter for the moment, thereby keeping the visual clutter to a minimum.

```
module Simpleex where

import Array -- standard Haskell array library

n = 3

data Fee = Foo | Fum deriving Show

f x | odd x  = Foo
    | even x = Fum

a = array (1,n) [(i,f i)| i <- [1..n]]
```

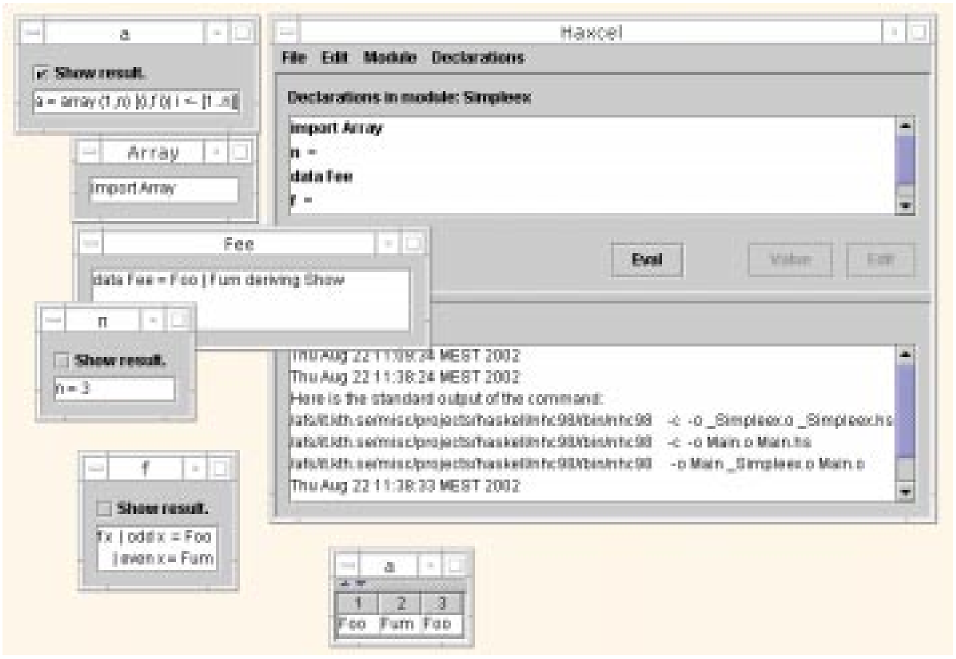
Fig. 2. Simple example module.

Having a value window for each value-binding declaration avoids the problem of unwanted side-effects at updates that occurs in traditional spreadsheets, where all tables and definitions reside in a single grid.

Most values are displayed with their standard `show` function. Arrays of dimension one and two can however be displayed as tables: one-dimensional arrays as linear tables, and two-dimensional arrays as matrices. See Fig. 3. Haxcel does not have any knowledge of types: therefore, the information needed to display arrays correctly (dimensionality, index ranges) is captured from the standard printout. Obviously, this mechanism will break if the `show` function for arrays is redefined.

Some values should not be printed, for instance if their types do not belong to the `Show` class, or if they are very large or infinite data structures. An attempt to show a value whose type has no `show` function will give a runtime error. It is up to the user avoid showing "harmful" values. A value is shown only if a "show value" box in the edit window is checked.

Haxcel also has a console window. Any messages from the compiler are shown there, including error messages. Haxcel itself does not perform any kind of syntax control or type checking of the definitions: all error handling is left to the compiler. In the case of an error, any current values in the open value windows are kept.



```
a = array (1,n) [(i,f i) | i <- [1..n]]
```

```
import Array
```

```
data Fee = Foo | Fum deriving Show
```

```
n = 3
```

```
f x | odd x = Foo
    | even x = Fum
```

Declarations in module: Simpleex

```
import Array
n =
data Fee
f =
```

```
Thu Aug 22 11:09:24 MEST 2002
Thu Aug 22 11:38:24 MEST 2002
Here is the standard output of the command:
/afs/it.kth.se/misc/projects/haskell/nhc98/bin/nhc98    -c -o _Simpleex.o _Simpleex.hs
/afs/it.kth.se/misc/projects/haskell/nhc98/bin/nhc98    -c -o Main.o Main.hs
/afs/it.kth.se/misc/projects/haskell/nhc98/bin/nhc98    -o Main _Simpleex.o Main.o
Thu Aug 22 11:38:33 MEST 2002
```

1	2	3
Foo	Fum	Foo

Fig. 3. Screenshot of Haxcel for the simple example module. All editing windows are open, plus the value window for a.

Haxcel saves its information about a module and its declarations in an internal format, with XML tags identifying the different properties. This is for two reasons: to save non-Haskell information about the GUI state, and to provide for easy identification of the declarations. In this way, no general parsing of the actual Haskell code is needed. An example of the internal format is shown in Fig. 4. Haxcel can export Haskell files, but currently it cannot import existing Haskell code.

4 Implementation

We designed Haxcel with the objectives in mind that it should be portable, and independent of Haskell compiler. Furthermore, we had limited time to complete the implementation. Thus, we opted for a design where Haxcel simply assembles the given declarations as they are into a module file, creates a Main module that imports the module, and whose `main` function prints the values of the identifiers with open value windows, ships the modules to the compiler for compilation, reads back the results from running `main`, and displays them in the proper windows. The compiler is invoked via `hmake`. As mentioned, any other output from the compiler is displayed in the console window.

This design has the advantage that the GUI becomes fully independent from any particular compiler. On the other hand, the turnaround time might be long, especially if large source files have to be compiled: see Section 7 for some experimental results. It is also hard to give the GUI any kind of syntactical or semantical knowledge of the

```
<?xml version='1.0' encoding='utf-8'?>
<module>
<modid name="Simpleex"/>
<declarations><declaration deftype="import">
<windowdata ewrect="398,107,147,71"
ewvisible="1"
/>
<haskell haskellname="Array" valuetype=""> <![CDATA[import Ar-
ray]]></haskell>
</declaration>
<declaration deftype="eq">
<windowdata ewrect="370,239,126,94"
ewvisible="1"
/>
<haskell haskellname="n" valuetype=""> <![CDATA[n = 3]]></haskell>
</declaration>
<declaration deftype="data">
<windowdata ewrect="399,179,280,99"
ewvisible="1"
/>
<haskell haskellname="Fee" valuetype=""> <![CDATA[data Fee = Foo | Fum de-
riving Show]]></haskell>
</declaration>
<declaration deftype="eq">
<windowdata ewrect="403,355,126,111"
ewvisible="1"
/>
<haskell haskellname="f" valuetype=""> <![CDATA[f x  | odd x  = Foo
    | even x = Fum]]></haskell>
</declaration>
<declaration deftype="eq">
<windowdata ewrect="354,14,192,94"
ewvisible="1"
vwrect="599,431,113,79"
vwvisible="1"
/>
<haskell haskellname="a" valuetype="array"> <![CDATA[a = ar-
ray (1,n) [(i,f i)| i <- [1..n]]]]></haskell>
</declaration>
</declarations>
</module>
```

Fig. 4. Internal format for simple example module.

code it processes: either this has to be implemented into the GUI itself, or somehow the needed information has to be got from the compiler. The second approach will make the GUI compiler dependent, since there are no standards for communicating type information and similar from Haskell compilers. Also the first approach will probably lead to some degree of incompatibility with some Haskell compilers: if a type checker is implemented in the GUI, for instance, then it will most likely not be compliant with all the extensions of Haskell's type system that the different existing compilers implement. Our design allows Haxcel to be used with any Haskell compiler while offering at least some basic spreadsheet functionality.

Later versions of Haxcel might provide running modes that interface closer to certain Haskell implementations. This could give increased ability to report on types, handle compilation errors, etc., for those implementations. Our current version of Haxcel has a mode where it uses `runhugs` to run the Hugs interpreter in batch mode, in order to get faster response times. See Section 7.

Haxcel is written in java. This is for reasons of portability, and also since there are many java libraries around for GUI development. The current implementation requires java v. 1.3.1 or higher, preferrably v. 1.4.

The current version of Haxcel is available on the web[1]. It has been tested with `ghc` v. 5.00.2 and `nhc98` v. 1.12 under Mac OS X, and with `nhc98` v. 1.14 under Solaris 2.6.

5 The Extended Array Library

The extended array library, `Xarray`, is designed to embed high-level array-language style programming into Haskell, with the purpose to support convenient array programming rather than highly efficient array computing. It supports the kind of array language features found in languages like Fortran 90 [2], including lifted numerical operations, constant arrays, various operations to split, compose and select parts of arrays, and a number of folds. The extended arrays provided by the library are essentially a simplified version of the *data fields* [16, 17] of Data Field Haskell [13], stripped down to be Haskell 98-compliant. The library can be used independently of the spreadsheet interface. All ordinary array operations in Haskell are provided for the extended arrays, so the library is backwards compatible. A list of the major new operations and type declarations provided by `Xarray` is found in Fig. 5. For their exact semantics, see the source code `Xarray.hs` which is available from the Haxcel home page.

An extended array is either an ordinary Haskell array or an *infinite array*. An infinite array is a function together with a particular bound `Universe`. Infinite arrays make sense in a lazy language, where finite pieces of them can be evaluated when needed. One particular reason to have them is that it is natural to overload numerical constants as infinite, constant-valued arrays, see further below.

Bounds for extended arrays can be either pairs of index values (for ordinary arrays) or `Universe` (for infinite arrays). The bounds form a lattice[2] with `Universe` as top element, and binary operations `join` and `meet`. See Fig. 6.

[1] `http://www.mrtc.mdh.se/projects/Haxcel/`

[2] There is no explicit representation for the bottom element, though.

```
data (Ix a) => Array a b = Arr (A.Array a b) | Inf (a -> b)
                           deriving ()
data Bounds a = B (a,a) | Universe deriving Eq

meet :: Pord a => Bounds a -> Bounds a -> Bounds a
join :: Pord a => Bounds a -> Bounds a -> Bounds a
inBounds :: Ix a => Bounds a -> a -> Bool
mkarray :: Ix a => Bounds a -> (a -> b) -> Array a b
finarray :: Ix a => (a,a) -> (a -> b) -> Array a b
infarray :: Ix a => (a -> b) -> Array a b
at :: (Ix a, Pord a) => Array a b -> (a,a) -> Array a b
when :: Ix a => (a -> Bool) -> Array a b -> b -> Array a b
proj_xx :: (Ix a, Ix b) => Array (b,a) c -> b -> a -> c
proj_1x :: (Ix a, Ix b) => Array (b,a) c -> a -> Array b c
proj_21 :: (Ix a, Ix b) => Array (b,a) c -> Array (a,b) c
...(etc)...
ifA :: (Ix a, Pord a) => (a -> Bool) -> Array a b -> Array a b
                                     -> b -> Array a b
cols2matrix :: Ix a => [Array a b] -> Array (a,Int) b
rows2matrix :: Ix a => [Array a b] -> Array (Int,a) b
matrix2rows :: (Ix a, Ix b) => Array (b,a) c -> [Array a c]
matrix2cols :: (Ix a, Ix b) => Array (b,a) c -> [Array b c]
foldlA :: Ix a => (b -> c -> b) -> b -> Array a c -> b
foldlM :: Ix a => (b -> c -> b) -> b -> Array a (Maybe c) -> b
...(etc)...
```

Fig. 5. Selected declarations and defined functions of the extended array library. (Haskell's usual `Array` module is imported qualified as `A`. `Pord` is a partial order class with operations `glb`, `lub`, and `lt` used by `join` and `meet`.)

`join` and `meet` are mostly used to compute bounds for arrays resulting from lifted, elementwise applied operations. For an operation that is strict in all arguments, the bounds for the array resulting from an application of the lifted version will be the `meet` of the bounds of the array arguments. `join` is used to compute the bounds for arrays obtained from lifted conditionals. This "datafield semantics" [16, 17] for elementwise applied operations comes from the view of arrays as partial functions, where the bound of an array represents (a possible overapproximation of) the domain of its partial function.

The extended array type `Array a b` is defined as an instance of `Num` and `Fractional` when `b` is an instance of `Num` and `Fractional`, respectively. The arithmetic operations on arrays are the lifted, elementwise applied operations on the element type. `fromRational` and `fromInteger` map numerical constants into infinite constant arrays. This overloading provides some common array language syntax. In particular, the datafield semantics for strict lifted operations makes it possible to have constants "promoted" to arrays of the right size when used as arguments to such operations. This works since infinite arrays have bound `Universe`, and the `meet` of `Universe` and `b` equals `b` for any bound `b`. For instance, scaling the array `x` with the factor `0.2` is simply expressed as `0.2*x`.

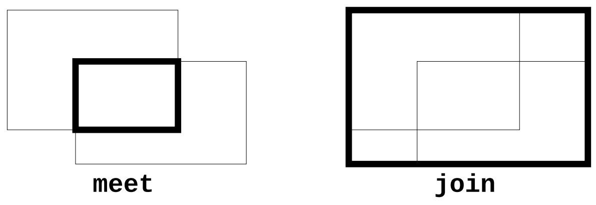


Fig. 6. `meet` and `join` on array (matrix) bounds.

Sometimes it makes sense to have sparse arrays in spreadsheet calculations, even though `Xarray` does not provide sparse array representations. The undefined elements must then be represented in some way, since we may want to perform operations like summation over a sparse array without having an error when an undefined element is encountered. This can be solved by using a `Maybe` type for the array elements, where `Nothing` represents undefined elements. In order to provide convenient array operations also on such arrays, `Maybe a` is defined as an instance of `Num` or `Fractional` when `a` is an instance of the corresponding class.

`Xarray` provides an elementwise conditional `ifA`, which takes a predicate over indices as condition and selects elements from two argument arrays depending on the condition. The bound for the resulting array is the `join` of the argument array bounds. This may yield an array where some elements are not given by the argument arrays (see Fig. 6): a fourth argument specifies which value to use then. Often, it is natural to use `Nothing`, since these elements in a sense are undefined, but sometimes it can be convenient to keep a non-`Maybe` element data type and use some value of that type.

`Xarray` defines two *restriction operations* on arrays. These operations select parts of arrays without changing the number of dimensions. `at` restricts with an array bound: `a ‘at‘ b` is the subarray of `a` with bounds `(B b) ‘meet‘ (bounds a)`. Thus, `a ‘at‘ ((2,2),(m,n))` selects the $(2, m) \times (2, n)$-submatrix of `a`. The other restriction, `when`, takes a predicate over indices and an array as arguments, and fills the array with a third argument in the positions where the predicate is false. This can be thought of as a masking operation. It does not change the array bounds. For instance,

```
when (i -> a!i /= Just 0) a Nothing
```

masks out the elements `i` where `a!i` is nonzero.

`Xarray` also provides a large number of *projection operations*. These operations select various subplanes of arrays, in different directions, and can also swap dimensions. A projection in a dimension means to fix the coordinate for that dimension. Projections can be made in several dimensions: if all dimensions are projected, then only a single array element remains. The unprojected dimensions are then mapped to the dimensions in the resulting array in a way specified by the particular projection.

The syntax of the projection function names is `proj_`c^n, where the symbols matching c are either “`x`” or a distinct number between 1 and the number m of non-`x`-symbols in the string. `proj_`c^n maps an n-dimensional array onto an m-dimensional array. An `x` in a certain position means that this dimension is projected, and a number that the dimension is mapped to the dimension with the given number in the resulting array.

For instance, `proj_1x a j` selects column `j` from matrix `a`, `proj_x1 a i` row `i`, `proj_12 a` returns `a` itself, and `proj_21 a` the transpose of `a`. `Xarray` provides all projection operations on arrays up to dimension three. See Fig. 7 for an example.

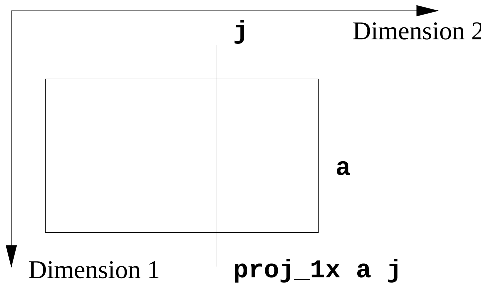


Fig. 7. Projection that produces a column from a matrix.

It is sometimes convenient to assemble arrays from arrays of lower dimensions, for instance to form a matrix from a number or vectors used as rows or columns. It is also sometimes useful to be able to decompose an array into a list of subarrays. `Xarray` provides operations to assemble matrices from vectors and split matrices into lists of vectors: `cols2matrix` and `rows2matrix` build matrices from lists of vectors as columns and rows, respectively, and `matrix2cols` and `matrix2rows` provide the inverse operations.

Finally, `Xarray` has a number of folds over arrays. They are all analogous with the usual folds for lists. There are two families: the first consists of “strict” folds that take all elements of the array into account (given that the folded operation is strict), and the second ignores array elements with the value `Nothing`. The second family is thus intended for “sparse” arrays. The folds in the first family are named by appending “`A`” to the name of the corresponding list fold, and those in the second family by appending “`M`”.

6 A Simple Example

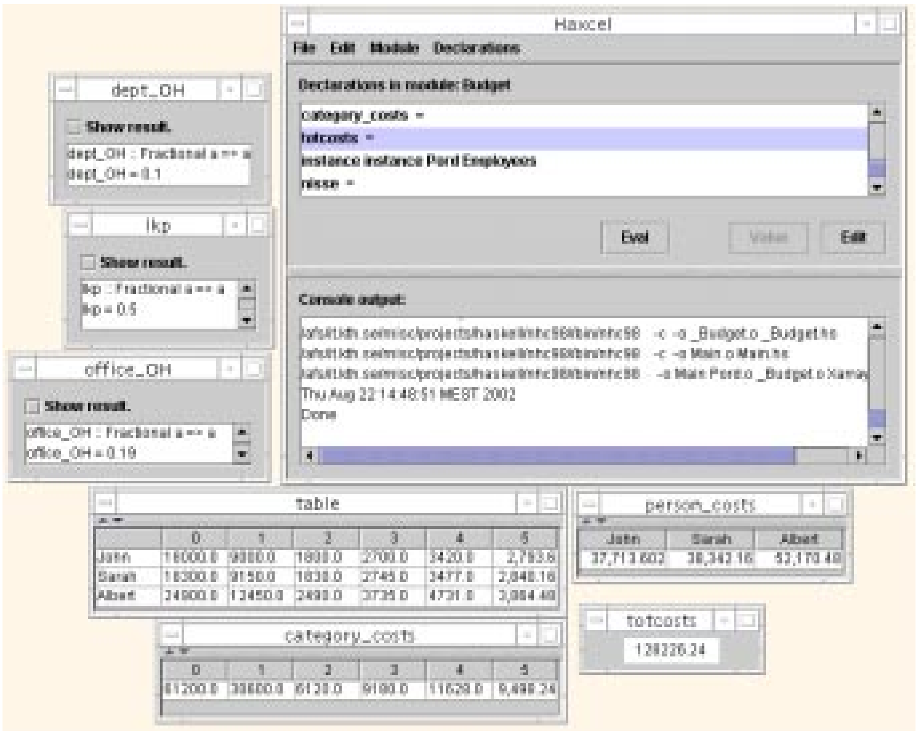
As a simple example of spreadsheet programming in Haskell with the extended array library, we give a little cost budget for a fictious research group. This group has three members. The costs are the salary costs plus a number of overheads, most of which are calculated as certain percentages of the salary cost, the VAT however as a fraction of the total cost (including the other overheads). We want to be able to maintain the budget as smoothly as possible: the members in the group might change, new costs may be charged the group, or the existing overhead ratios may change. We want to calculate the total cost, and the cost per person, but also the cost broken down in different categories (department overheads etc.).

Our solution is to put all costs in a matrix, whose first dimension is indexed by a type with one constructor per member. The matrix is assembled from a number of vectors, one per cost category. To keep overhead ratios easy to modify they are made symbolic through simple definitions, which are simple to change if the need arises. The definitions are given in Fig. 8, and a snapshot of the Haxcel screen is shown in Fig. 9. (This snapshot shows a hypothetical situation where we want to play around with the overhead factors in order to see the effects on the various parts of the budget. Thus, the editing windows for the overhead factors are open, as well as the value windows for the costs in the budget.)

```
module Budget where

import Ix

import Xarray

data Employees = John | Sarah | Albert
                 deriving (Eq,Ord,Ix,Show,Enum,Bounded)
instance Pord Employees

salaries :: Fractional a => Array Employees a
salaries = listArray (minBound,maxBound) [18000.0,18300.0,24900.0]

-- Overheads:
lkp, dept_OH, univ_OH, office_OH, vat :: Fractional a => a

lkp = 0.5
dept_OH = 0.1
univ_OH = 0.15
office_OH = 0.19
vat = 0.08

table = let l = [salaries,
                 lkp*salaries,
                 dept_OH*salaries,
                 univ_OH*salaries,
                 office_OH*salaries]
        in cols2matrix (l ++ [(sum l)*vat])

-- sum of costs for each employee
person_costs = sum (matrix2cols table)

-- sum of costs for each cost category
category_costs = sum (matrix2rows table)

-- total cost
totcosts = sumA person_costs
```

Fig. 8. Haskell code for the budget example.

Note the explicit typing `Fractional a => a` on the “scalar” numerical values `lkp` etc. They are used to scale arrays. Without this typing they would not be lifted to infinite arrays, and the scaling wouldn’t work. This problem is due to the monomorphism restriction in Haskell’s type system.



Declarations in module: Budget

```
category_costs =
totcosts =
instance instance Pord Employees
nisse =
```

```
dept_OH :: Fractional a => a
dept_OH = 0.1
```

```
lkp :: Fractional a => a
lkp = 0.5
```

```
office_OH :: Fractional a => a
office_OH = 0.19
```

Console output:

```
/afs/it.kth.se/misc/projects/haskell/nhc98/bin/nhc98  -c -o _Budget.o _Budget.hs
/afs/it.kth.se/misc/projects/haskell/nhc98/bin/nhc98  -c -o Main.o Main.hs
/afs/it.kth.se/misc/projects/haskell/nhc98/bin/nhc98  -o Main Pord.o _Budget.o Xarray
Thu Aug 22 14:48:51 MEST 2002
Done
```

	0	1	2	3	4	5
John	18000.0	9000.0	1800.0	2700.0	3420.0	2,793.6
Sarah	18300.0	9150.0	1830.0	2745.0	3477.0	2,840.16
Albert	24900.0	12450.0	2490.0	3735.0	4731.0	3,864.48

John	Sarah	Albert
37,713.602	38,342.16	52,170.48

0	1	2	3	4	5
61200.0	30600.0	6120.0	9180.0	11628.0	9,498.24

128226.24

Fig. 9. Snapshot of the Haxcel screen for the budget spreadsheet.

Our simple example demonstrates some features that might be useful when programming spreadsheet applications. Ordinary spreadsheets can be heterogenous, and strings are often put into cells to tell what numerical fields, rows, or columns mean. In Haskell, one can instead use simple algebraic data types like `Employees` to create descriptive index sets for arrays. An advantage, besides the type-safety, is that several arrays now can have their index sets given by the same data type. This is a win since one often uses different arrays to describe different properties of the same set of items (like the employees of the research group in our example). If this set changes (if, say, we hire more people), then, if the arrays are defined in the right way, it will be very easy to update the spreadsheet by simply changing the data type declaration and adding/removing input data in the proper arrays. In the usual spreadsheet model, this kind of update can be very error-prone.

It is not hard to imagine an interactive mechanism for defining arrays, where index sets can be created and copied between arrays, and the corresponding data type declarations are automatically generated.

7 Experimental results

To get a grip on the turnaround time for evaluating spreadsheets, we did some simple benchmarking. We ran the two examples presented here, `simpleex` (Section 3) and `budget` (Section 6). We measured: the time to assemble the Haskell files for compilation, the time for hmake/nhc98 to compile, the time for the generated program to execute, and the time to parse and display the results. We ran the examples on two machines: a Power Mac G4 with a 400 MHz PowerPC G4 processor, 192 MByte RAM, and 1 MB cache, under Mac OS X 10.1.5 with the nhc98 v. 1.12 compiler, and a Sun Ultra Enterprise 4000/5000 with six 167 MHz Sparc processors (whereof we used one), 256 MB RAM and 0.5 MB cache per processor, under Solaris 2.6 with the nhc98 v. 1.14 compiler. On the Power Mac we also ran both examples with runhugs rather than hmake. Each example was run 5 times for each configuration. The results (mean values) are presented in Table 1. Note that the different times were measured with different resolution, due to the kind of timing tools we had available: the java-specific parts (assemble, parse & display) were measured with resolution 1s, and the compilation and execution were both measured with resolution 0.01s on the Mac and 0.1s on the Sun, respectively. The variation in measured time between runs was low, except for the measurements of the java-specific parts where the low resolution sometimes caused jumps in the observed times.

`Simpleex`					
	Assemble	**Compile**	**Execute**	**Display**	**Total**
Mac/`runhugs`	0.0	N/A	0.60	0.0	0.60
Mac/`hmake`	0.0	5.44	0.06	0.0	5.50
Sun	0.2	7.4	0.0	0.0	8.2

`Budget`					
	Assemble	**Compile**	**Execute**	**Display**	**Total**
Mac/`runhugs`	0.0	N/A	0.79	0.4	1.19
Mac/`hmake`	0.0	7.37	0.29	0.4	8.06
Sun	0.2	10.3	0.7	0.4	11.6

Table 1. Measurements for `Simpleex` and `Budget`, in seconds.

Not surprisingly, the main time for Haxcel with hmake/nhc98 is spent compiling. This yields a turnaround time of the order 10s on our systems. This is not readily acceptable for an interactive environment. On the other hand, better compilation times might be obtained on other platforms: for instance, Wallace [24] claim compilation times for `nhc98` in the order 1-2 seconds for medium-sized modules, on a PC with Linux and a 500 MHz pentium processor.

For Haxcel with runhugs on the Power Mac, we obtain much faster turnaround times, of the order 1s for both examples. Although not instantaneous, this is a much more acceptable response time.

Most of the display time for `Budget` can be attributed to the first run for each experiment, where extra time is spent opening the value windows. Subsequent runs mostly have lower display times.

8 Related Work

The first spreeadsheet application was VisiCalc [3] for Apple II. Since then, a myriad of different commercial and experimental spreadsheet systems have been designed and implemented. Here, we review the work on spreadsheets with functional formula languages, and spreadsheets that allow the kind of non-grid-confined cell concept that our interface implements.

There have been many attempts to desing spreadsheet environments with a full functional programming language as formula language. On Linux, there are several spreadsheet systems available that use some interpreter for Scheme: an example is SIAG [10]. Davie and Hammond considered "functional hypersheets" based on Haskell [7], but we do not know whether their design was ever implemented. De Hoon et. al. describe a spreadsheet imlemented in Clean, which has a simple, list-based, untyped functional formula language "FunSheet" with a rewrite semantics [8]. FunSheet uses a conventional spreadsheet indexing scheme with letters for columns and numbers for rows. However, columns are considered functions, so the function application `A 1` refers to cell `A1` in conventional spreadsheet syntax. This indexing scheme is elegant, but from an array processing point of view it has the disadvantage that it favours accesses by columns whereas many matrix and table calculations require equally convenient access to rows or other array subsections. An interesting feature of FunSheet is a limited capability for symbolic processing, which is made possible by the rewrite semantics.

Clack and Braine describe a spreadsheet design based on the object-oriented functional language CLOVER [6]. Their ideas to use higher order techniques for flexible spreadsheet programming are similar to ours. Mini-SP, by Yoder and Cohn [26], is an APL-like spreadsheet programming language, which has a concurrent evaluation model.

Spreadsheets with cells but without a grid have also been considered. An early example is Nas, a simple visual language with this property [25]. Nas was however more of an example of an advanced functional programming application, and was primarily used to evaluate implementation techniques for lazy functional languages. Forms/3 [4] is an attempt to create a spreadsheet programming environment for graphical applications: it has a first-order recursive functional formula language and free placements of cells.

What we believe sets our work apart from other spreadsheet environments is the following: Haskell as formula language, the view of the spreadsheet interface as a component that can use different compilers as "evaluation engines", and the extended array library that combines array language functionality with higher order programming techniques in a spreadsheet setting.

On the Haskell side, Haxcel is related to interactive Haskell interfaces such as Hugs [14], GHCi [22, Ch. 3], and in particular hi [24]. In comparison with these, Haxcel provides a less text-oriented interface that can help visualize results of program changes faster.

The array-oriented programming style supported by `Xarray` is found in array- and data parallel languages such as Fortran 90 and HPF [15], and *lisp [23]. These languages are collection-oriented – a very good survey of such languages is found in [21]. It is also interesting to compare with functional array languages, notably the classical data flow languages Id [9] and Sisal [11], as well as the later, Sisal-inspired SAC [19]. SAC is especially interesting since it has an array model that is inspired by APL. SAC programs can have a high degree of *dimension-invariance*, i.e., the same program works with arrays of any dimensionality. These functional languages emphasise parallelisation and speed of compiled code, however, and therefore have limitations in the array syntax and the overall generality. For instance, they are all first-order and they do not have full type-inference.

There are some other array extensions of Haskell. Chakravarty et. al. [5] have extended Haskell with nested data-parallel arrays, much in the style of NESL [1]. Hill [12] extended Haskell with lazy, unbounded data parallel structures that can be seen as a kind of sparse arrays. Both these attempts focus on expressing parallelism and achieving efficiency of implementation, rather than on arrays as a high-level concept for tasks like spreadsheet calculations. Our own Data Field Haskell [13] is different in this respect, in that it implements an extended array model that aims primarily at expressiveness and swift denotation of array-like computations. It would not be hard to extend Haxcel with support for Data Field Haskell.

9 Conclusions and Further Research

We have designed Haxcel, a compiler-independent spreadsheet interface for Haskell programming, and the accompanying extended array library `Xarray`. Together, they demonstrate what a high-level programming environment can look like, which can be used both for conventional Haskell programming and for spreadheet calculations. We also think Haxcel overcomes some of the problems with the conventional spreadsheet model, and it extends current spreadsheet languages to the full power of Haskell.

Clearly, many features can be added to Haxcel. The ability to import existing Haskell modules has not been implemented only due to lack of time. Better means to define arrays interactively would be nice for spreadsheet programmers: currently, they can be defined only through textual Haskell expressions. Better graphical support to display data structures like lists is desirable, as well as the possibility to link cells to external viewers or players, and an ability to import data from the outside and convert into suitable Haskell values. The simple textual input of declarations could be enhanced into a more visual programming environment. The current turnaround times for small examples are annoying for hmake/nch98 but acceptable when using runhugs. If still faster turnaround times are needed, then one could add a mode where hugs runs in a parallel thread rather than being invoked in batch mode. If this is still not enough, then one could make a more elaborate dependency analysis to recalculate and update only those values that have changed since the last evaluation. We believe all these features would be fairly straightforward to implement.

The current lack of error handling in Haxcel is not satisfactory, but it is dictated by our wish to keep Haxcel as a standalone GUI component that is compatible with Haskell compilers in general. Also, the lack of knowledge about types is a limitation. As mentioned earlier, a possible solution is to add modes to Haxcel where it interfaces more closely to certain compilers. However, this complication could be avoided if there were standard protocols for Haskell compilers to communicate type information and error messages. Such protocols could then also be used by other Haskell tools. Also, if parsers in Haskell compilers could be used as components to produce parsed code in some standard format, then tools like Haxcel could take advantage of them. The proposed Haskell Execution Platform [20] has addressed some of these issues, but we don't know to which extent it has materialised.

Do we think that Haskell is a good language for spreadsheet calculations? Yes, probably for someone who has a good grasp of modern functional programming. But we do not think Haskell in its current form is the perfect spreadsheet language. Spreadsheet calculations are best expressed in an array model, and while quite a few array language primitives can be reasonably well embedded in Haskell we have also found some annoying limitations. For instance, the ability to overload numerical literals as infinite constant arrays, but having to provide explicit type annotations for variables defined to have the values of these literals, is a safe bet for tripping spreadsheet programmers not well acquainted with Haskell's type system. Furthermore, Haskell was designed with lists rather than arrays in mind: thus, all the "good" operators and function names are already taken for list operations. A good spreadsheet language, which suits the causal user, should have obvious names for important functions over arrays, and it should not require explicit type declarations for common spreadsheet programming. Some of these problems could possibly be alleviated by providing a "simple spreadsheet mode" to Haxcel, where assumptions about data types are made and explicit type declarations are automatically generated, but this feature remains to be added and evaluated.

The large number of projection operations that we had to define is also a nuisance. Array languages often have intuitive syntax to express these things: for instance, Fortran 90 allows expressions like `A(3,2:N,:)` to select from `A` the submatrix with the first coordinate fixed to `3`, the second varying in the range `2:N`, and the third in `A`'s extents in the third dimension. The `forall` statement of Data Field Haskell [13] can also express these kinds of projections conveniently. However, such features cannot be added to Haskell without a change in syntax. Thus, maybe the best solution for a spreadsheet language would be to make a derivative of Haskell which is more oriented towards high-level array processing.

References

1. G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *J. Parallel Distrib. Comput.*, 21(1):4–14, Apr. 1994.
2. W. S. Brainerd, C. H. Goldberg, and J. C. Adams. *Programmer's Guide to FORTRAN 90*. Programming Languages. McGraw-Hill, 1990.
3. D. Bricklin and B. Frankston. VisiCalc: Information from its creators, 1999. `http://www.bricklin.com/visicalc.htm`.
4. M. Burnett, J. Atwood, R. W. Djang, J. Reichwein, H. Gottfried, and S. Yang. Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. *Journal of functional programming*, 11(2):155–206, Mar. 2001.
5. M. M. T. Chakravarty, G. Keller, R. Lechtchinsky, and W. Pfannenstiel. Nepal — nested data parallelism in Haskell. In R. Sakellariou, J. Keane, J. R. Gurd, and L. Freeman, editors, *Proc. 7th International Euro-Par Conference*, volume 2150 of *Lecture Notes in Comput. Sci.*, pages 524–534. Springer-Verlag, 2001.

6. C. Clack and L. Braine. Object-oriented functional spreadsheets. In *Proceedings of the 10th Glasgow Workshop on Functional Programming*, GlaFP'97, September 1997.
7. A. Davie and K. Hammond. Functional hypersheets. In *Proc. 8th Int. Workshop on Implementation of Functional Languages*, pages 39–48, 1996.
8. W. A. C. A. J. de Hoon, L. M. W. J. Rutten, and M. C. J. D. van Eekelen. Implementing a functional spreadsheet in Clean. *Journal of Functional Programming*, 5(3):383–414, 1995.
9. K. Ekanadham. A perspective on Id. In B. K. Szymanski, editor, *Parallel Functional Languages and Compilers*, chapter 6, pages 197–253. Addison-Wesley, 1991.
10. U. Eriksson. Siag office - a free office package for unix, Mar. 2002. `http://www.siag.nu/`.
11. J. T. Feo, D. C. Cann, and R. R. Oldehoeft. A report on the Sisal language project. *J. Parallel Distrib. Comput.*, 10:349–366, 1990.
12. J. M. D. Hill. *Data-Parallel Lazy Functional Programming*. PhD thesis, Department of Computer Science, Queen Mary & Westfield College, University of London, Sept. 1994.
13. J. Holmerin and B. Lisper. Data Field Haskell. In G. Hutton, editor, *Proc. Fourth Haskell Workshop*, pages 106–117, Montreal, Canada, Sept. 2000.
14. M. P. Jones and J. C. Peterson. Hugs 98 user manual, Sept. 1999. `http://www.haskell.org/Hugs/downloads/hugs.pdf`.
15. C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele, Jr., and M. E. Zosel. *The High Performance Fortran Handbook*. Scientific and Engineering Computation. MIT Press, Cambridge, MA, 1994.
16. B. Lisper. Data fields. In *Proc. Workshop on Generic Programming*, Marstrand, Sweden, June 1998. `http://wsinwp01.win.tue.nl:1234/WGPProceedings/`.
17. B. Lisper and P. Hammarlund. The data field model. Technical report, Department of Computer Engineering, Mälardalen University, Västerås, June 2001. `http://www.mrtc.mdh.se/publ.php3?id=0312`.
18. J. Malmström. A spreadsheet interface to Haskell. Technical report, Department of Computer Engineering, Mälardalen University, Västerås, 2002. Forthcoming.
19. S.-B. Scholz. On Programming Scientific Applications in SAC - A Functional Language Extended by a Subsystem for High-Level Array Operations. In W. Kluge, editor, *Implementation of Functional Languages, 8th International Workshop, Bad Godesberg, Germany, September 1996, Selected Papers*, volume 1268 of *Lecture Notes in Comput. Sci.*, pages 85–104. Springer-Verlag, 1997.
20. J. Seward, S. Marlow, A. Gill, S. Finne, and S. P. Jones. Architecture of the Haskell Execution Platform (HEP). Technical report, Microsoft Research, Cambridge, UK, July 1999.
21. J. M. Sipelstein and G. E. Blelloch. Collection-oriented languages. *Proc. IEEE*, 79(4):504–523, Apr. 1991.
22. The GHC Team. The Glasgow Haskell Compiler user's guide, version 5.02, 2002. `http://www.haskell.org/ghc/docs/latest/set/book-users-guide.html`.
23. Thinking Machines Corporation, Cambridge, MA. *Getting Started in *Lisp*, June 1991.
24. M. Wallace. hi — hmake interactive — compiler or interpreter? In M. Mohnen and P. Koopman, editors, *Proceedings of the 12th International Workshop on Implementation of Functional Languages*, Aachener Informatik Berichte, pages 339–345. RWTH Aachen, 2000.
25. S. C. Wray and J. Fairbairn. Non-strict languages — programming and implementation. *The Computer Journal*, 32(2):142–151, Apr. 1989.
26. A. G. Yoder and D. L. Cohn. Real spreadsheets for real programmers. In *International Conference on Computer Languages*, pages 20–30. IEEE, 1994.