A User-Centred Approach to Functions in Excel

30th June 2003

Simon Peyton Jones
Microsoft Research, Cambridge
simonpj@microsoft.com

Alan Blackwell
Cambridge University
alan.blackwell@cl.cam.ac.uk

Margaret Burnett
Oregon State University
burnett@cs.orst.edu

Abstract

We describe extensions to the Excel spreadsheet that integrate user-defined functions into the spreadsheet grid, rather than treating them as a "bolt-on". Our first objective was to bring the benefits of additional programming language features to a system that is often not recognised as a programming language. Second, in a project involving the evolution of a well-established language, compatibility with previous versions is a major issue, and maintaining this compatibility was our second objective. Third and most important, the commercial success of spreadsheets is largely due to the fact that many people find them more usable than programming languages for programming-like tasks. Thus, our third objective (with resulting constraints) was to maintain this usability advantage.

Simply making Excel more like a conventional programming language would not meet these objectives and constraints. We have therefore taken an approach to our design work that emphasises the cognitive requirements of the user as a primary design criterion. The analytic approach that we demonstrate in this project is based on recent developments in the study of programming usability, including the Cognitive Dimensions of Notations and the Attention Investment model of abstraction use. We believe that this approach is also applicable to the design and extension of other programming languages and environments.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language classifications—*Functional languages*; D.3.3 [**Programming Languages**]: Language constructs and features—*Procedures, functions and subroutines*; H.1.2 [**Models and principles**]: User/machine systems

General Terms

Languages,Design

To appear at the International Conference on Functional Programming (ICFP'03), Uppsala, 2003

1 Introduction

For many people, the programming language of choice is a spreadsheet. This is especially true of people who are not employed as programmers, but write programs for their own use — often defined as "end-user" programmers [Nar93]. An end-user programmer is a teacher, an engineer, a physicist, a secretary, an accountant, a manager, in fact almost anything except a trained programmer. These people use computers to get their job done, but often they are not interested in programming *per se*. End-user programmers outnumber professional programmers, and their numbers are projected to increase more rapidly; in fact, the number of end-user programmers in the U.S. alone is expected to reach 55 million by 2005, as compared to only 2.75 million professional programmers [BAB+00]. These facts suggest that the spreadsheet, which is a widely used and commercially successful end-user programming language, is also a particularly significant target for the broader application of programming-language design principles.

It may seem odd to describe a spreadsheet as a programming language. Indeed, one of the great merits of spreadsheets is that users need not think of themselves as doing "programming", let alone functional programming — rather, they simply "write formulae" or "build a model". However, one can imagine printing the cells of a spreadsheet in textual form, like this:

```
A1 = 3
A2 = A1-32
A3 = A2 * 5/9
```

and then it plainly is a (functional) program. Thought of as a programming language, though, a spreadsheet is a very strange one. In particular, *it is completely flat*: there are no functions apart from the built-in ones. Instead, the entire program is a single flat collection of equations of the form "variable = formula". Amazingly, end users nevertheless use spreadsheets to build extremely elaborate models, with many thousands of cells and formulae. Almost all of these elaborate models are constructed in part by reusing formulae from some cells in other cells via the clever spreadsheet copy-and-paste techniques.

From a programming language point of view, then, *spreadsheets lack the most fundamental mechanism that we use to control complexity: the ability to define re-usable abstractions*. In effect, they deny to end-user programmers the most powerful weapon in our armory. Can you imagine programming in C without procedures, however clever the editor's copy-and-paste technology?

In this paper, we describe a possible extension to Excel that supports user-defined functions (Section 4). It turns out that support for functions is closely associated with better support for vectors and matrices as first-class values, and we discuss that too (Section 5). The basic idea is simple. We propose that functions be defined in

terms of the formulae that are contained in a specified part of the spreadsheet. The function signature is defined by allocating some subset of the cells as function parameters.

The main contribution of our work is not the user-defined function idea *per se*, but the large amount of detailed design required to integrate this new view of spreadsheet behaviour into an *existing* product whose continued survival depends on its genuine usefulness to humans.

Because we focus on end users rather than professional programmers, usability is a fundamental concern, so we been consciously guided by research into human-computer interaction. Beyond its specific design, our work shows how principles drawn from HCI research can be applied to the design of programming environments, in a very concrete, practical way. This is important – the ultimate goal of all programming technology is to make people more productive – but other documented examples are hard to find.

We hope that the paper may also serve to remind the functional programming community of a large but mostly-ignored group of functional programmers; namely, end users. Highly-expressive programming is particularly important for this group, so functional languages are especially relevant.

We are in active discussion with the Excel developers about the possibility of including user-defined functions in a future version of Excel. Our progress in these discussions has been encouraging, but whether we will ultimately succeed in that endeavour is, of course, uncertain.

2 A user-centred approach to language design

Despite the deficiencies of spreadsheets when they are considered as programming languages, it is clear from their commercial success that people find them useful and usable. This is not true to the same extent of any traditional programming language. Why not? One contributing factor may be that, although programming languages are a channel of communication between people and computers, most programming-language research is devoted to the computer end of the channel. Comparatively little research considers the people who do the programming [NC85]. It is our belief that programming language design projects should adopt a more user-centered approach, in order to maintain a sharp focus on the people who are intended to benefit from the new language features or environments being created. This is the approach that we have adopted in our own work on extending spreadsheet capabilities.

The groundwork for this project comes from specialised research into the psychology of programming and empirical studies of programmers (e.g. [BG87, CW00, DP95, GPB91, HLMC96]). It is hard, though, for a programming-language designer to find practically useful guidance from such research. Individual studies in these fields are often experimental investigations of relatively constrained contexts or points of syntax, so that it can be hard to find empirical evidence that usefully informs design decisions for new languages. Furthermore their theoretical generalizations tend to emphasise cognitive theories of problem solving, which are not readily applicable by programming language designers.

To address this problem, HCI researchers have developed structured approaches to considering human issues in programming. We single out two — Cognitive Dimensions and Attention Investment — that have been used successfully in other recent programming language and environment design projects (e.g., [LGAH02, BCACA02, BBC02]). In the following sub-sections we briefly review these two frameworks.

Abstraction gradient	What are the minimum and maximum levels of abstraction? Can fragments be encapsulated?
Consistency	When some of the language has been learnt, how much of the rest can be inferred?
Error-proneness	Does the design of the notation induce 'careless mistakes'?
Hidden dependencies	Is every dependency overtly indicated in both directions? Is the indication perceptual or only symbolic?
Premature commitment	Do programmers have to make decisions before they have the information they need?
Progressive evaluation	Can a partially-complete program be executed to obtain feedback on "How am I doing"?
Role-expressiveness	Can the reader see how each component of a program relates to the whole?
Viscosity	How much effort is required to perform a single change?
Visibility and juxtaposability	Is every part of the code simultaneously visible (assuming a large enough display), or is it at least possible to compare any two parts side-by-side at will? If the code is dispersed, is it at least possible to know in what order to read it?

Figure 1. Cognitive Dimensions (excerpted from [GP96])

2.1 Cognitive Dimensions

The *Cognitive Dimensions of Notations* (CDs) framework distills knowledge from the psychology of programming into a form that can help non-psychologists to assess usability at design time [GP96, BG03]. CDs are not language design rules, but instead provide *vocabulary* that enumerates concepts important to human problem solvers who are engaged in programming tasks. Many of the CDs express concepts that are already very familiar to language designers, but in an informal way. As a result, they may seem obvious, but informally familiar design concerns are easily forgotten when attempting to maximize other properties such as conciseness. The benefit of CDs lies in having an explicit list of design attributes — summarizing empirical evidence about programming language or environment attributes important to human problem solving — which can be checked and referred to throughout a design project. Our own design was guided by our use of the CDs framework during the design process.

An example of a CD is *consistency*. Although most language designers know generally that consistency is a "good thing", the consistency dimension sharpens the concept by expressing it as a question: "When some of the language has been learnt, how much of the rest can be inferred?" [GP96]. Expressed in this way, the consistency dimension makes a point that is particularly relevant to our project, because it is a key to end users' likelihood of successfully *using* user-defined functions in spreadsheets.

Another example is *viscosity*: the amount of effort required to make a small change to the information structure. For example, a language that lacks procedural or functional abstraction is highly viscous, because code must be repeated. This means that changing the code later takes work, because all of its repeated locations must be changed.

Figure 1 summarises the CDs that we use in this paper. We use them in a *formative* (before-the-fact) way, to inform our design while it is

taking shape, by raising warnings when a design decision is potentially at odds with research from cognitive principles of programming. Of course, Cognitive Dimensions cannot "prove" that no such problems remain. Eventually, complementary devices aimed at *summative* (after-the-fact) evaluation, such as empirical studies, are also needed.

2.2 Attention Investment

The Attention Investment model of abstraction use analyses the actions that will be taken by a programmer in terms of investments of attentional effort [Bla02]. Here is the idea. Programmers have a finite supply of time to spend concentrating on their work, or *attention*, to *invest*. They will invest their effort in activities for which their expected *payoff* exceeds the *cost*, unless the *risk* is too great. The *cost* of the investment is the amount of attention by the user that must be devoted to accomplishing a task. The expected *payoff* from that investment will be some saving of attentional effort in the future, such as by achieving a good abstract formulation to reduce the amount of effort required to cope with similar problems. The perceived *risk* is the extent to which the user believes the investment will not produce the payoff, or that it will lead to even more costs that are not yet apparent.

This simple investment model, when applied at multiple levels of granularity, can model many of the actions and decisions made during programming tasks. The model has been validated in a cognitive simulation of programmer behaviour for fine-grain decisions [Bla02], and there is evidence that it is effective in practical language design [BB02, BBC02].

As with the CDs framework, most language designers have an intuitive understanding of these issues. The model's value for our work has been in making these psychological concerns explicit, so that we can analyse and justify design decisions. As an intellectual tool for language designers, a strength of the Attention Investment model is its generality, which allows it to provide insights not produced by the more specific CD framework. On the other hand, the concrete enumeration of issues in CDs is more prescriptive of specific problems to consider, and provides sharper questions for language designers. For this reason, we consider the two approaches to be complementary intellectual tools, and use them both in the work described here.

3 The user-centred case for functions

Our goal is to extend Excel by making it easier for end users to define and re-use their own libraries of domain-specific functions. Is this a good idea in the first place? After all, spreadsheets have evolved over a good number of years, so maybe the status quo is near-optimum. In this section we make the case for user-defined functions, using the language of Attention Investment to consider potential payoffs and costs from a human productivity standpoint.

3.1 What are the payoffs?

Users will invest effort if they foresee a reasonable payoff. First we must consider the potential *actual* payoffs. Whether our target audience will *perceive* these payoffs will be discussed in the relevant sections that follow.

For programming language folk it is not too hard to see what the actual payoffs of user-defined functions might be:

Avoid repetition. Suppose a user repeatedly types a formula such as $IF(A6 = 0, "", A6)$, which blanks out zero values. This repeated work is tiresome. Even worse, if instead of $A6$ the user wants a formula such as $SUM(D1{:}E9) * 2$, it would have to be duplicated in the IF formula. User-defined functions enable them to identify, name, and re-use code that they use again and again: thus $BLANK\_ZERO(A6)$, or $BLANK\_ZERO(SUM(D1{:}E9) * 2)$.

Reduce errors during maintenance. Excel encourages copy-and-paste of complex formulae; but if a copied formula must be modified later, all the places it was copied before must be found and updated, which is *error-prone* (see Figure 1). A named function encapsulates the formula, thereby protecting against that source of risk. It also reduces the user's attentional cost by reducing *viscosity* (resistance to change), because the encapsulated formula needs to be changed in only one place. Our interview with an auditor at a major accounting firm revealed that he considers the problems of error proneness to be quite severe: "I will remind you that in 6 years work, checking literally hundreds of business-critical models, ... my team have never failed to find errors."

Real estate management. A function call takes up only one cell to invoke, even if the function's definition spans many cells. This provides a natural way to save real estate on the invoking sheet. Real estate management is clearly important in practice: Excel has a great deal of support for it (hiding rows and columns, collapsing groups, and so on).

Encapsulate and re-use domain-specific expertise. User-defined functions directly support re-use. Local domain experts can create libraries of functions, tailored to their particular application and context, that can be re-used by their colleagues. Our interview with another customer revealed the high potential he ascribed to this benefit: "If we can crack the ability to reuse blocks of code it will be immensely valuable, both within a spreadsheet to curb repetition, and between spreadsheets too in the form of standard libraries."

Intellectual property protection. While function bodies can be manifest, they do not have to be. User-defined functions provide an easy way to encapsulate and distribute intellectual property.

Performance. Behind the scenes, a function can be represented as an expression tree, compiled to byte code, JITted to machine code, or whatever, with substantial performance benefits. Interviews have revealed that for certain important user groups (notably in finance) performance is critical.

More generally, the goal is simply to provide end users with the same complexity-management tools that we take for granted as professional programmers.

3.2 What are the costs?

User-defined functions are, in fact, already available in Excel; all you need do is define your function in Visual Basic[1]! The trouble

[1] Earlier versions of Excel supported user-defined functions via so-called "XLM macros", not to be confused with what Excel currently calls a "macro". XLM macros looked superficially more familiar than Visual Basic to a spreadsheet user, because the macro code was laid out on a grid of cells. The semantics of the grid was rather different to an ordinary worksheet: it was in fact an imperative programming language, with sequential evaluation and a program counter, but with Excel's formula language as a sub-language. The attention-investment cost was still high, albeit not as high as learning Visual Basic. In fact, XLM macros are still available (right-click on a worksheet tab, select "Insert..." and pick "Excel 4.0 macro sheet"), but they are not documented or encouraged.

is that the cost (in terms of attention investment) is very high. The programming paradigm is different (imperative instead of declarative); the notation is different (block of text instead of a grid of cells); the programming environment is different and complex (Visual Studio); the debugging model is different and less accessible (debugging in Excel means looking at values that are continuously displayed in intermediate cells, while debugging in Visual Basic means managing the Visual Studio debugger). These differences lead to high learning costs.

One of our meetings with Excel users contributed further insights into additional costs in the real world: "Excel already has the opportunity to package up oft-repeated calculations as Visual Basic functions. However, what you are proposing is much to be preferred... One [advantage] is performance; VB functions can be rather slow. ... VB functions break the audit trail; not all of their behaviour is determined by their parameters, as they can retrieve data from cells other than through the parameter list. Debugging VB functions requires programming skills; yours requires more standard spreadsheeting skills." With costs like these it is hardly surprising that many users never make the investment.

3.3 Costs vs payoffs: our target audience

Our design lowers the total costs, but not to zero. In order to understand the costs of our design, it is first necessary to identify which of the learning cost elements are zero, i.e., the *prerequisite* skill set expected of the intended audience [YBDZ97]. A prerequisite is both an asset and a barrier — designers can build design features on the prerequisites, but people who do not have the prerequisites should not be expected to succeed at tasks that require those prerequisites. For our design, the prerequisites are these: users should be (1) comfortable with using a variety of built-in functions, not just the infix operators; (2) comfortable using Excel's copy/paste or replicate operations, in which a formula is systematically altered to suit its new location; and (3) able to use more than one worksheet in a workbook[2]. The first and the third items provide the scaffolding upon which our approach rests. The second item is tied with increasing the attention payoff, because it is in this item where the long-term maintenance costs of these users' current procedures can be reduced.

Some people use Excel for nothing more than managing and printing a list of information. Others know how to use very simple formulae, such as $=SUM(A1{:}A10)$, but nothing more. Our design has little to offer these groups, because the payoff (approximately zero) is lower than the learning costs of the prerequisites.

Our primary target audience is moderate users — those who understand the spreadsheet paradigm fairly thoroughly. Not only have they already mastered the prerequisites, but they also tackle more ambitious and long-lived applications than does the first group, so the payoff is greater.

Advanced users can also benefit. Advanced users are those who understand Visual Basic and use it to write functions they call from Excel formulae. Our design will be valuable for them if it enables them to accomplish, at lower cost or lower risk, some tasks that previously required Visual Basic.

[2]In Excel, a file contains a *workbook*; a workbook consists of one or more *worksheets*, each of which has up to 256 columns and an arbitrary number of rows. Using multiple worksheets makes it easier to avoid logically separate computations "bumping into each other".

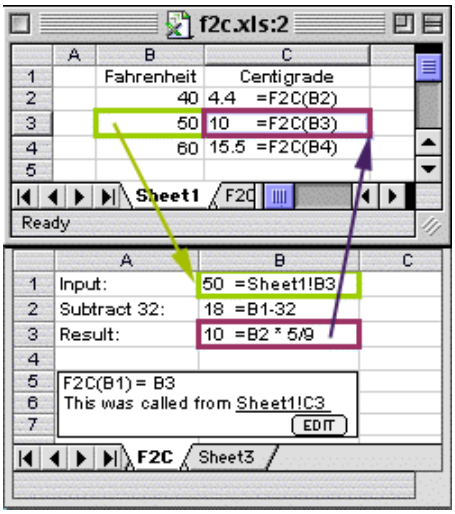

Figure 2. A function instance sheet (bottom) and its invocation site (top)

4 User-defined functions in Excel

How, then, should user-defined functions be added to Excel? In this section, we show how the CDs and the model of attention investment revealed the answers to a number of language design issues that arose in attempting to answer this question.

4.1 Applying the consistency CD

Recall from Figure 1, the *consistency* CD is expressed as "When some of the language has been learnt, how much of the rest can be inferred?" Since we would like our target audience, whose prerequisite skills we enumerated in Section 3.3, to be able to infer (have a low-cost way of learning) how to create, view, and modify, a user-defined function, applying the consistency CD is critical. We applied it by establishing the following ground rule, which is the key distinguishing feature of our design, relative to any commercially-deployed alternatives. (We discuss related work in Section 6.)

> *The implementation of a function must be defined by a spreadsheet, because that is the only computational paradigm understood by our target audience.*

In keeping with this ground rule, in our design a function implementation is given by a worksheet, which we call a *function instance sheet*, or sometimes just "function instance" for short. As the prerequisites list of Section 3.2 has already established, our target users bring to the table an understanding of the idea of multiple worksheets, and that worksheets can be *linked* by writing a formula in one worksheet that refers to a cell in another. To be consistent with these prerequisites, our design makes a function sheet look and behave much like a linked worksheet.

Figure 2 shows an example. The upper worksheet is an ordinary worksheet. Cell C3 contains an invocation of a user-defined function $F2C$, which converts temperatures in Fahrenheit to Centigrade. The implementation of $F2C$ is given by a function instance sheet, shown below in the same figure[3]. The input parameter lands in cell $B1$ of the $F2C$ function instance; cell $B2$ contains an intermediate value, while the final result is computed in $B3$. The fact that that $B1$ contains the input and $B3$ the result is also specified in the *interface panel* that appears at the bottom of every function instance, which

[3]Excel already has facilities to show multiple worksheets one above the other.

defines the external interface to the function. Notice, too, that the input cell $B1$ contains the formula $= Sheet1!B3$, which makes explicit where the function instance gets its input from, using Excel's existing notation for cross-sheet links. Cells $A1 - A3$ contain commentary written by the user.

From the figure it is clear that user-defined functions are like spreadsheets in appearance. The consistency CD makes clear that they must evaluate like spreadsheets too. Thus, all values are continuously calculated and displayed. If the user changes the value in $B3$ of the calling sheet, not only is the new result of the invocation calculated and displayed in $C3$, but the values in each cell of the $F2C$ sheet are also updated with their new values. Similarly, any change to the $F2C$ sheet is immediately reflected by re-calculation.

To maintain consistency with other worksheets, function instance sheets can also contain formatting, funny column widths, fancy borders, charts, and whatever, all of which may help someone looking at the internals of the function. In short, an underlying principle behind every design choice has been consistency in every way possible between function instance sheets and other worksheets.

This consistency helps to align the users' *perceived* costs and risks of functions with the *actual* costs and risks. For example, a function sheet's immediate recalculation sends a visual message to the user, that these sheets are like the ordinary worksheets to which the user is accustomed.

For consistency with built-in functions, a user-defined function has exactly the same status as a built-in function, once defined. For example, Excel's existing "function-entry wizard" (a little f_x button to the left of the formula bar) pops up a menu of possible functions, organised by category; when one is chosen, a second dialogue pops up with a box for each parameter, each with its name, and an accompanying summary of what the function does. In our design, all of this works the same with user-defined functions as with the built-ins.

What does the consistency CD imply about *recursive* functions, so dear to the hearts of functional programmers? While recursion still makes sense in the spreadsheet paradigm [BAD+01], it is much less useful than in mainstream functional languages: in the absence of algebraic data types, the only inductive type is integers — and one can iterate over integers using arrays, much as in APL (Section 5)[4]. Further, recursion would threaten consistency with the linked-worksheet model, because the number of linked worksheets would be data-dependent. Lastly, recursion leads to deeper invocation stacks, which translates into more function sheets potentially cluttering up the workspace, each element of which adds very little new information to the human problem solver.

In short, recursion threatens consistency and potentially adds attention-investment costs without adding corresponding payoffs. For these reasons, in our design we chose not to support recursive user-defined functions[5]. This decision was not taken lightly, but is a conscious prioritisation of usability over generality.

4.2 Visibility issues

Role expressiveness and *hidden dependencies* are two CDs whose influences on our design choices can be seen in Figure 2. Excel already does a nice job of expressing each cell's role in a calculation with its color-coded rectangles; we extend this technique to make clear the roles of actual/formal parameters and the return value. Hidden dependencies ("Is every dependency overtly indicated in *both* directions?") are something of an issue in the commercial version of Excel in the presence of linked worksheets, because the dataflow arrows work only for references within the same spreadsheet. In our design, dataflow arrows can connect linked worksheets, as shown in the figure. As the figure shows, these two devices make explicit the role of the new function relative to the other worksheets.

The *visibility* CD emphasizes the importance of every part of the program being simultaneously visible (assuming a large enough display). The side-by-side aspect of *juxtaposability* (discussed in [GP96]) also brings out the importance to human problem solvers of being able to display related items side-by-side. Since cells' values and their formulae are clearly related, in our design it is possible to view formulae and values together, side by side, as shown in Figure 2. Further, following from the visibility aspect of this CD, in our design we allow display of formulae and values simultaneously in *all* visible cells, not just one cell at a time. (Excel currently allows viewing of all cells' values with only one cell's formula, or all cells' formulae with only one cell's value, and the formula is displayed in a separate window pane away from the values.)

4.3 Applying the premature commitment CD

In programming languages for professional programmers, programmers (1) first realize they want to define a function, (2) then they define that function, and (3) finally they invoke the function. Step (1) must occur before step (2), which must occur before step (3).

But our target audience does not necessarily possess the knowledge that tells them they should define a user-defined function — recall there is nothing in their prerequisites list about prior experience with user-defined functions. Any design that required the above sequence would be ignoring the *premature commitment* CD, which warns about requiring an audience to make a decision before they have the information to do so. As this CD makes clear, since the above sequence is unlikely to be followed by our target audience, our design must instead support a sequence that does not require such premature commitment. Therefore, the sequence we support is this: (1) the user has already written some code (formulae) that works well, (2) the user decides the code needs to be reused or repeated, (3) the user discovers a low-cost low-risk way to reuse the existing code through user-defined functions, and (4) the user acts upon the discovery by converting their existing code to a user-defined function, which is automatically invoked, when sensible, where the existing code used to be.

[4] Excel also supports another, dangerous form of iteration, via circular formulae (e.g. A1 = A1 + 1). This sort of "recursion" utterly breaks Excel's basic functional paradigm, because each recalculation sweep computes a fresh iteration, so *the value of each cell changes over time*. Our proposals do not make this technique safer, nor do they intefere with it.

[5] Can the user create a recursive function by accident? As in all other formula edits, if the user attempts to enter a recursive, or even mutually recursive formula, the system must attempt to compute the answer. In the process of doing so, if the system re-invokes the same function name, it can stop computing and reject the newly entered formula.

To support this sequence, we allow the user to start with the code they have written in these two ways:

- Start from a formula in a single cell, or
- Start from an existing ordinary worksheet range of cells.

Note that neither case requires the user to know in advance that she is going to create a function; rather we provide ways for her to take her work so far and encapsulate it.

To create a function from a formula in a single cell, the user brings up the cell's property menu (with a right-click) and selects "Make a function using this formula". This action creates a new function sheet, displayed in a second window. Thus, the user's actual cost is low: a right-click to pop up the property menu and select the option. (Users will encounter this option when they have other reasons to pop up the cell's property menu.) Their risk of losing their invested effort in creating the previous formula is also low, since the answer in the cell remains the same. The wording of this menu option has been deliberately chosen to be non-destructive in an attempt to communicate this low risk, and the fact that there are no ellipses or sub-menus attempts to communicate the low cost.

In creating the new function, the system automatically guesses the number and order of parameters based on the formula, and replaces the original formula by an invocation of the new function, whose name is provisionally just "UNTITLED". The number and order of parameters is necessarily a guess; for example, in the formula $(A6 - 32) * 9/5$ are there four parameters (*A*6, 32, 9, and 5), three, two, or just one (*A*6)? Or perhaps there is just one parameter $(A6 - 32)$. The system can make a reasonable guess, but there is no way to be sure. In any case, the user can readily fix up the guess by editing the interface (Section 4.6).

The other way to create a function is from a larger portion of an existing worksheet. To do so, the user selects a *range* of cells, right-clicks, and selects "Make a copy into a function". The original range is left unchanged, but is also copied into a new function sheet. Again, the system guesses inputs and outputs, this time based on dependencies in the range, and the user edits the result. Initially, there is no invocation of the new function.

The consistency CD arises again here, this time in opposition to concepts from Attention Investment. Clearly it is not consistent that our design for generating a function from a range of cells makes a copy, leaves the original unchanged, and does not invoke the new function – whereas generating a function from a single cell actually replaces the original formula with a call to the new function. However, the user's perception of cost and of risk could both be adversely affected if an entire range of cells' formulae were to disappear and be replaced with a single call to another sheet. Further, the likelihood of correctness of the system's guesses of inputs and outputs is lower, which means the user would have to patch up cell relationships without having the original version with which to compare. (While an "undo" button could rescue the user from such functions that seem wrong, it cannot make the *use* of functions seem less costly — instead, it simply makes it easy for the user to decide not to go forward with functions.) In essence, the cost of consistency is too high in this case, in terms of both the user's attention cost and their possible perception of the risk involved in using functions.

4.4 Applying the abstraction gradient CD

In programming languages we are accustomed to distinguishing between a function *definition* (which we see in our editor) and a function *invocation* (which we only see in our debugger, as the transient contents of a stack frame). A function definition is abstract, in the sense that it does not include any actual data from a specific instance of invocation. This means that, when writing a function, programmers have to model (in their heads or on paper) the actual data that will be present when the function is invoked. This modeling process is an additional attention cost [Bla02], and causes problems for end users working with functions in other languages. These cognitive challenges are described by the CD of *abstraction gradient*.

In contrast, Figure 2 involves no definition/invocation distinction. There, the *F*2*C* function is shown not as an abstract definition, but as an actual instance of the function being invoked. This function instance is populated with "live" data originating from a specific invocation site — the cell where the function is called (the name of the actual invoking cell is in the interface panel). If there are many invocations to *F*2*C* then, conceptually at least, there are many copies of the *F*2*C* worksheet, each with the same layout and formulae, but populated with different data. There is no such thing as the "definition sheet" for a function; every function instance is populated with data from somewhere.

This means that the user can immediately see the effect of any change on some set of concrete data. This feature is analogous to the object-oriented prototype/instance model, in which the functional equivalent of a class is defined via a prototype object, from which additional copies (instances) can be made. Perhaps the best known language following this model is Self [US87]. The prototype/instance paradigm, like our function instances, reduces the user's attention cost by reducing the abstraction gradient.

The user's experience of these function instances is very similar to that of linked worksheets, bringing the benefits of a low learning cost from this consistency, but it raises some interesting design questions:

- If there are thousands of invocations of a function, how can we avoid overwhelming the user with thousands of worksheets (Section 4.5)?
- If only function instances are displayed, how can one edit the function definition (Section 4.6)?
- What are the implications for debugging functions (Section 4.7)?

4.5 Managing visibility costs

Our design greatly increases the amount and kind of logic visible in worksheets, and with that increased visibility comes the increased attention cost of managing the display. For example, when each function instance is displayed as a worksheet, there may be a great many worksheets active at once. How can the user see just the ones she wants?

Excel can display multiple *windows* simultaneously, inside the overall Excel frame. In each window, Excel displays one *worksheet* at a time. The user can select which worksheet is displayed in a window by clicking on that sheet's *named tab*. For example, Figure 2 shows a 2-window display, in each of which a different sheet has been selected.

The obvious model is to have a named tab for each function instance (since they are meant to be like worksheets), but that leads to two immediate problems. First, there are just too many of them; with a bit of copy-and-paste, it is easy to create thousands of function invocations. Second, and more subtly, it is hard to *name* the function instances — in concrete terms, what should we display on the function instance's tab to identify it? Clearly the name of the function alone is not unique; nor is the name together with the call site, because the call site's sheet will have the same naming problem if it too is a function instance.

We must provide low-cost ways for the user to navigate this sea

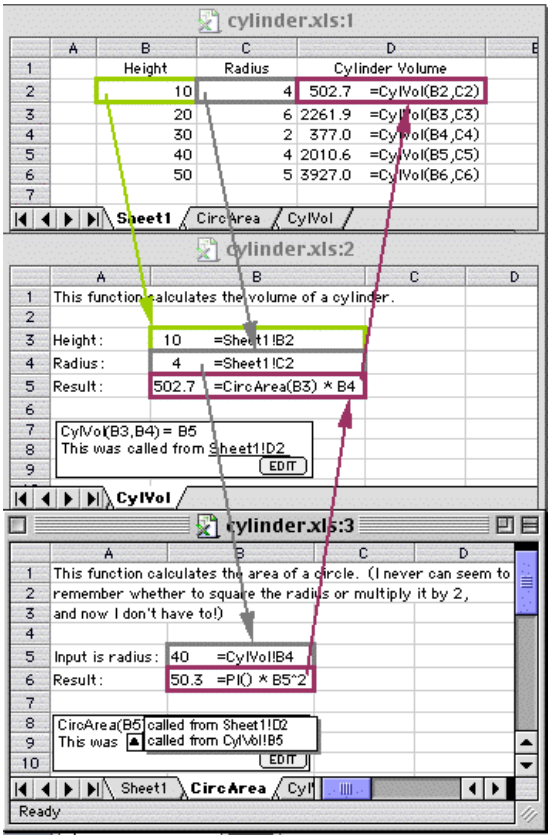

Figure 3. A function that calls another function

of function instances. We do so by providing, for every function instance, (a) a direct "link" from a cell to the function instance(s) invoked by the formula in that cell, (b) a direct link from a function instance to the formula that invoked it, and (c) a direct connection between instances of the same function. More specifically, given an *invocation tree* (or call tree) of function instances, the user can navigate in the following ways.

Navigating down the tree. Suppose that a cell (on any sheet – either an ordinary sheet or a function instance) has a formula containing an invocation of a function `Foo`. The user selects the cell and requests[6] the function instance sheet for a call in that cell; in response, the sheet appears below the invoking sheet. For example, if only the top sheet of Figure 2 had been visible, and the user then interacted in this way with cell *C*3, the screen would appear as shown in the figure.

Navigating up the tree. Given a function instance sheet, how does the user find where in the invocation tree that sheet is? That question is answered by looking at the function instance's summary interface panel, which is always displayed at the bottom of the sheet. It says explicitly where the function is invoked from; for example "Called from *A*3 of *Sheet*1", or "Called from *B*7 of function *Foo*". In the latter case, a popup-menu triangle appears next to "Called from", which brings up the entire stack of "Called from" specifications, all the way up to an ordinary function sheet. Selecting any member of this stack causes it to be displayed in a new window, now above the current one, thereby navigating up the invocation tree. Again following the local tree navigation metaphor, the immediate invoker can be displayed by a single click on the "Called from..." link. Figure 3 gives an example of nested function invocation, showing the popup menu popped up in the *CircArea* sheet. This menu also allows the user to locate a particular function instance in the invocation tree, alleviating the "lost in hyperspace" problem.

Excel's usual behaviour when one clicks on a tab is for the specified worksheet to *replace* the one currently displayed. When navigating the call tree, however, it is very desirable to show the user both *both* the function's definition *and* its context of use at the same time — the CD of *visibility/juxtaposability* again. In our design, therefore, when navigating down the tree, the current worksheet slides up the display, and the specified function instance is shown in a second window below it. Dually, when navigating up the tree, the current worksheet slides down, and the invoking sheet is displayed above it.

Function instances still have a tab, like other worksheets, but the tab displays only the name of the function. Clicking on the tab brings the most recently-viewed instance of that function to the front (without any window-shifting; this is just the normal click-on-tab behaviour). So, in effect, all the instances of a particular function share a single tab.

Jumping across the tree. What if the user wants to explore more than one part of the invocation tree at once? If those parts involve different functions, he can just click the tab of the appropriate function. If he wants to see them side-by-side, he can use Excel's fine existing support for juxtaposability by adding new windows.

If the two parts involve different instances of the *same* function, matters are not so clear, because there is only one tab for that function. We deal with this case by adding a drop-down menu to the function tab, which gives access to recently visited instances of the function, identified (albeit perhaps ambiguously[7]) by their immediate call site.

4.6 Applying the hidden dependencies CD

The *hidden dependencies* CD rises to particular importance when we consider the issue of change. This CD calls attention to the danger of unintended consequences when users make changes without full awareness of the dependencies involved. The hidden dependencies CD suggests that a way to head off such problems is to make dependencies explicit. In this section, we show how our design acts upon this suggestion in two change situations: changing a function's implementation and changing a function's interface.

First, consider changes to a function's *implementation*. Obviously, the way the user edits the function's implementation is simply to change the cells on one of the function's instances, but when happens to the other instances? (Recall, there is no "master copy" of the function; every worksheet is simply an instance, of equal stature as all the other instances.)

When one makes a recurrent appointment in an electronic diary, and then tries to modify it, a pop-up dialogue box shows up saying "Do you want to alter just this appointment, or the whole series?" In our design, function editing works in the same way. If the user tries to modify a function instance, a pop-up box asks whether she wants to modify all the invocations to the function in the same way (the default), or whether she wants to create a special-purpose copy of the function for this particular invocation. (If there is only one invocation of the function, a common case when developing a function, the "modify this one or all?" question is not asked at all.) Asking

[6] Via a button or right clicking and select "show sheet for `Foo`".

[7] The potential ambiguity is like the "Alt-Tab" display through which the Windows operating system offers rapid, but ambiguous, navigation to running applications.

the user explicitly whether she intends to make a global change, draws her attention to — and gives her control over — the dependencies impacted by the change.

If the user asks to modify all invocations of the function, the function instance becomes open especially for multiple-instance editing (with visual feedback to that effect) so that the same question is not asked repeatedly, which would greatly raise attention costs. If the user asks to modify just the current invocation, the function is automatically renamed, because it is now a genuinely new function, distinct from the original.

Second, consider changes to a function's *interface* — for example, to add a parameter, or to change the name of the function. The user's mechanics are straightforward: the `Edit` button on the interface panel allows the interface to be edited, by bringing up a dialogue box that shows the complete interface (which includes the function's name, information about each parameter, devices to add or remove parameters, documentation and help text, etc.) However, how the system should respond if the function has existing invocations is not always so straightforward.

If the user has changed the function's name, or the order of its parameters, the system's response is simply to make the same change at each invocation site. Deleting a parameter can work the same way. Adding a parameter is more problematic because heuristics would be needed, and any wrong guesses could "invisibly" introduce bugs without the user realizing the extent of changes being made at multiple locations, including locations not visible on the screen. Even one such occurrence could greatly increase the user's costs and perception of risk. To avoid this problem, in the case of adding parameters, the invocation sites are flagged as "broken" (e.g. `Foo(A2, 7)` becomes `Foo(A2,??,7)`). For this approach to be useful, broken instances must be easy for the user to find. Excel has a way to select[8], and then visit, cells with certain properties (holding an error value, holding a boolean formula, etc). In our design, the same technique is used to find cells with broken invocations, and also cells that invoke a particular function. Any of these invocations that do not get fixed — that invoke non-existent functions or have the wrong number of parameters — will result in error values, which are Excel's equivalent of raising exceptions. Such exceptions explicitly inform the user of dependencies requiring attention due to the interface changes.

4.7 Applying the progressive evaluation CD

In other programming languages, a particular function invocation may exist only fleetingly during the execution of the program. Considerable planning (and its corresponding attention cost) may be required to halt the program in a state that exhibits the fault; and even that often reveals merely that a different (often earlier) program state must be captured.

The CD known as *progressive evaluation* points the way to a solution that avoids these costs, namely to automatically reflect incrementally the specific values in each visible function instance resulting from every formula edit. The progressive evaluation CD describes this incrementality, and research shows that it is of critical importance to novice users [GP96]. Thus, in our design, debugging functions is, like any other kind of debugging in spreadsheets, an activity that can begin as soon as the code-building process begins, while the program is still incomplete. Even half-complete functions contain live data that is continuously kept up to date.

Obviously, this is quite unlike conventional debuggers: there is no notion of "stepping forward" or "setting breakpoints"; the programming model is simply a static tree of linked worksheets, all simultaneously active and up-to-date. For example, if the user should wonder why the invocation $F2C(50)$ gives the result 10, he can look at the intermediate values that are always displayed in the $F2C$ instance for that call, as in Figure 2.

As this example illustrates, in our design, the live data in a function instance is *persistent*[9]. Indeed, the entire invocation tree is persistent, so that the user can view any number of function instances in the tree simultaneously. (That is why we generally use the term "invocation" rather than "call"; it avoids the here-today-and-gone-tomorrow implications of the latter term.) For example, Figure 3 shows a three-window view, showing the entire invocation tree for a small program. The absence of recursion facilitates navigation of the invocation tree: (a) its structure does not depend on the input data, and (b) repetition is expressed through matrix operations (Section 5) which do not clutter the invocation tree, rather than through recursion, which would.

Debugging functional programs has received quite a bit of attention from the programming-language community, both for strict languages [Lie84, FFF+97, TA90] and lazy ones [HO85, Nil98, SR97]. Some of these proceed by working backwards from an erroneous result (algorithmic debugging); others work forward by allowing the user to "step into" redexes that she identifies. All make strenuous efforts to present program fragments in their original source form. Our debugging model has a very different flavour to any of these, because there is no mode change between writing or modifying a program, and debugging it. Changes to the source are immediately reflected in changes of the execution. Every intermediate value is readily accessible, surrounded with any comments, related values, or other context, that the programmer originally wrote. Overall, the integration of debugging with programming is so complete that users may not even regard it as a separate activity. This integration helps to align the perceived costs of working with user-defined functions with the actual costs.

The use of progressive evaluation does not mean that spreadsheets are any more correct than other programs! Much research (surveyed in [Pan98]), has shown that spreadsheets are typically riddled with errors, and users wildly over-estimate the reliability of their spreadsheet models. Approaches have been proposed to help address these problems [RLDB98, IMCZ98, Dav96], and these approaches are directly applicable in the context of Excel-with-functions.

5 Matrix values in spreadsheets

It may not seem obvious that there is any need for a matrix type in a spreadsheet. The spreadsheet itself is very clearly a visual representation of a matrix, and the conventional way of using spreadsheets is to divide the plane into individual patches, each holding a one- or two-dimensional collection of data values.

Functions change the picture, though, as the *consistency* CD makes clear. The issue is consistency of user-defined functions with built-ins. Consider the built-in *SUM* function. It takes a vector or matrix as its argument; thus $SUM(A1{:}A10)$ or $SUM(A1{:}D10)$. Since built-in functions in Excel already take matrices as arguments, user-defined ones ought to do so too.

The next question is how. Consider a user-defined function to compute the sum of the squares of the elements of a vector. Where on the function instance sheet should the input vector reside? It could take up the first row, perhaps, but (a) we do not want to specify in the function's definition the size of the vector, (because the built-ins

[8] This facility is not well known: Edit/GoTo.../Special..

[9] Persistent in the user model, that is. The implementation can, of course, cache or recompute function instance data on demand using any caching strategy.

accept matrices of arbitrary size), and (b) we might like a function to work on matrices and there is only room for one unbounded matrix on a worksheet!

The obvious solution is to allow an input vector (or matrix) to live in a single cell. Values in Excel carry dynamic type information – for example, strings, numbers and error values are distinguishable. All that is required is to add an extra run-time type, namely *matrix*. We deliberately use terminology that is semantically oriented ("vector", "matrix") rather than implementation-oriented ("array"). Excel already supports matrix values in so-called "array formulae", but that support relies upon explicit size information in every reference to and propagation of the array of values, which we have already pointed out violates consistency with built-in functions. Our design extends Excel by making matrices truly first-class citizens that support consistency.

The idea of matrices as first-class values, together with a rich library of functions that operate over such values, was first popularised by APL, but has been widely used since in languages as diverse as High-Performance Fortran and Haskell, among many others. The trick, of course, is how to incorporate them smoothly into Excel in a way that will be actually used by our target audience of "moderate" users (Section 3.3).

5.1 The ground rules

The two cognitive dimensions of consistency and error proneness, along with the Attention Investment notions of cost and risk, led to our basic design choices:

- *Consistency*: Any formula can have a matrix as its value.
- *Consistency*: Matrices are two-dimensional, in keeping with Excel's two-dimensional paradigm. (However, a matrix element can be of any type, including a matrix, and this offers much of the power of N-dimensional matrices.)
- *Consistency*: A vector is just a special case of a matrix (1xN or Nx1). This means that horizontal vectors are not the same as vertical vectors, which is consistent with users' experience of working with rows and columns in spreadsheets.
- *Error proneness*, *cost*, *risk*: Because of Excel's default of treating empty cells as zero-valued cells for some functions, there are likely to be subtle differences between doing some kinds of matrix operations with a 1x1 matrix as versus with a constant. To avoid the risk of subtle, hard-to-find errors that could arise, we do not define a 1x1 matrix to be the same as a scalar. However, we also do not want to impose upon the user the seemingly arbitrary cost of turning scalars into 1x1 matrices. Thus, a scalar is implicitly promoted to be a 1x1 matrix in any context where a matrix is required. For example, consider a function *BESIDE* which combines two matrices side-by-side into a bigger matrix. The formula $BESIDE(1, B1{:}B2)$ works fine, even though 1 is not a matrix; it is simply promoted to a 1x1 matrix.

5.2 Reducing perceived costs and risks in working with matrices

The user is not required to have the time and inclination to go exploring for matrix features in order to eventually discover matrices. (Recall that interest in exploring new features was not one of the prerequisites we included in defining our target audience.) Rather, a key to our design is that matrix values are likely to be automatically created when the user starts creating his or her own functions. For example, if a cell contains the formula $= SUM(A1{:}A5)/4$, and the user uses the "Make a function from this formula" technique described in Section 4.3, then the newly-created function will automatically have a vector input parameter ($A1{:}A5$) landing in its input cell. This provides a low-cost entry point into the use of matrix values.

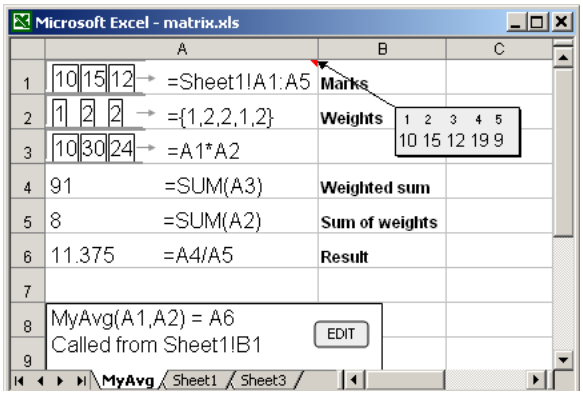

Figure 4. A worksheet that manipulates vectors.

Once this event has occurred, we would like the user's perception to be that the risks involved are low. Figure 4 will serve as an example to discuss the steps we have taken toward this goal.

Figure 4 shows a function *MyAvg* that might form part of a teacher's grading application. It takes a vector of marks (input cell $A1$) and a vector of weights (input cell $A2$), and computes the weighted sum of the marks (result cell $A6$). Cell $A3$ also contains a vector, the result of multiplying the marks by the weights element-wise. The formula $= SUM(A3)$ in cell $A4$ adds up the elements of the vector argument to *SUM*, namely the vector in $A3$. The invocation on $Sheet1$ (not shown) would look like $= MyAvg(A1:A5, \{1,2,2,1,2\})$, as can be seen from the formulae in the input cells of *MyAvg*. (The notation $\{1,2,2,1,2\}$ is Excel's existing notation for a literal vector value.)

A key to reducing perceived risk is to help the user understand the meaning of the current state. (Dropping the users into a state they do not understand is likely to increase their perception of risk, which could send them directly to the 'undo' button.) Thus, following the CD of *role expressiveness*, the roles of the matrix values in the function are explained to the user in two ways. First, the panel at the bottom shows the overall role of the matrix values in this function. Second, any cell that contains a matrix value, such as cells $A1, A2, A3$, gives a visual cue to its contents. Figure 4 gives an example of one possible cue, in which the cell contents shows the first few values, together with an arrowhead suggesting more values. Hovering the mouse over the cell brings up a scrollable floating panel, much like that for a comment, that allows the user to examine (but not modify) the value of the matrix. This, too, is shown in Figure 4.

5.3 A potential risk: trapped in a matrix

From a risk perspective, we do not want the gathering of a group of values into a matrix to mean that the propagations and results of these values can never again be spread out over a range of cells. For example, in Figure 4, the values of range $= Sheet1!A1{:}A5$ have been gathered into a single cell $= MyAvg!A1$. If the range had been quite large (e.g., $= Sheet1!A1{:}A100$), most values would not be visible, except by scrolling the floating panel. If it were not possible to translate the matrix value back to a range of cells, users would be restricted as to how they could view not only that matrix, but also all future matrix values propagated from the original. Since the uses of all future propagations cannot be predicted, this would be a clear

case of the CD *premature commitment*, requiring people to make decisions before they have all the necessary information.

To avoid this potential risk, we allow matrix values to be spread over a range of cells. More precisely, a range of cells may be given a single, matrix-valued formula, whose value is then spread across the range. For example *A*1:*A*10 contains a horizontal vector value. To spread out a value over a range of cells, our syntax is this: select a range of cells, such as *A*1:*A*10, and type a formula, such as $= B1{:}B12$. (This is the equivalent of the strictly textual syntax of $A1{:}A10 = B1{:}B12$.) The value denoted by this formula ($= B1{:}B12$) is then spread out over the selected cells (*A*1:*A*10). If the value is too small, the unused cells are blank; if it is too big, the excess values are not shown (and indeed are not accessible).

When hovering the mouse over a range of cells that share a single formula in this way, the range highlights. If the range is too small to fit the matrix value, the appropriate border(s) of the highlighted range are given a ragged edge to indicate the non-visible values. One can then drag a handle in the corner to resize the range, thereby showing more or fewer elements of the matrix.

6 Related work

6.1 Other approaches to functions in spreadsheets

Excel today provides user-defined functions by allowing the user to write a function in Visual Basic. We will refer to this a "trapdoor" approach, because it is simply an exit from the usual language to a pre-existing one from the world of traditional programmers.

Several research systems have supported functions in spreadsheets. Except for the Forms family (discussed next), they are all either imperative or trapdoors. Examples of trapdoors that are also imperative include Penguims [Hud94], Action Graphics [HM90], SIV [CRB+98], Spreadsheet for Images [Lev94]. Prograph [SCB96] is similar except that the trapdoor goes the other way: one takes a trapdoor out of Prograph into a spreadsheet (with imperatives in it). C32 [Mye91] uses a semi-declarative approach via a trapdoor into Lisp. (By semi-declarative, we mean that it is declarative to the same extent that Lisp itself is.)

The notion of declarative functions as multiple sheets was first introduced by Forms [Amb87]. Although Forms itself was a short-lived language, its concept of functions as multiple sheets survived in the later members of this family of languages. The concept of how to support functions as multiple sheets was extensively developed in Forms/3 [BA94, BAD+01], and this language influenced our Excel extension.

However, research systems have the luxury of not having to support existing communities or real-world needs. In light of the realities introduced by these factors, the Excel extensions described here differ from previous research in a number of significant ways. First, previous systems did not address creating a function from a pre-existing formula or spreadsheet. Rather, the notion was that a user would explicitly define a function by building a spreadsheet from scratch. Defining functions from pre-existing formulae or sheets is critical for creating a gentle migration path, because it allows legacy Excel workbooks to be gradually transformed, one function at a time, as repeated code is discovered.

Second, previous research did not deal with some of the important scalability issues that related to usability, such as how to locate one of many functions and instances of the same functions. Scalability is often the critical point that determines whether solutions that look fine on toy academic problems can be truly useful. Because of its practical importance, this has been an important issue in the design of the Excel extension.

Finally, there is support in the Forms/3 approach to functions for some features that are specifically not included in the Excel extension. Two of the most noticeable such features are automatic generalization of function calls and support for recursion. Neither of these features is expected to be of importance to our target audience, and we have elected not to include them.

6.2 Other usability techniques

Empirical evaluation (or a "usability study") is a reasonably common approach to the evaluation of new user interface designs. Many research evaluations in HCI rely on controlled experiments, where user performance on an experimental task can be compared for alternative interfaces. The problem for programming language design is how to select suitably representative tasks, reflecting both the scale and variety of potential programs that might be created. Empirical studies that address such broad questions are generally too expensive to inform a design like ours during its evolution, although they have value in a summative (after-the-fact) role. We do intend to follow standard Microsoft practice in conducting usability studies of our design before it is brought to market. We consider that this is an important reality check for any innovation.

Empirically based design. A few researchers have followed an approach in which empirical studies were conducted before the main design, testing specific experimental hypotheses rather than general measures of utility. Examples of this approach include the Hands language [PMM02] and SWYN [Bla01]. The similarity to our work is philosophical — in their work and ours, the emphasis is on the human from the very outset of the design process. However, these projects do not involve existing languages intended for production use, and the experiments isolate factors of interest rather than evaluating their contribution to a larger system. This means that the research process can be rather simpler than the constraints we face. Nevertheless, as most such research is published, we have been able to take advantage of findings from experimental studies testing aspects of Attention Investment, Cognitive Dimensions, and novel spreadsheet systems.

Language design heuristics. A popular alternative to expensive empirical studies is Heuristic Evaluation [NM90]. Summaries of good practice and of results from previous research are systematically compared to a new design, in order to apply the lessons from things that worked well, or badly, in the past. Heuristic evaluation is not normally applied to programming languages, but an exception is the reference manual compiled by Pane and Myers collecting usability factors in programming language design [PM96]. They concentrated on factors that are relevant to novice programmers, mainly because this population is seen as suffering more severely when programming languages are hard to use.

The techniques briefly surveyed in this section concentrate on evaluating and improving specific features of a language or programming environment. In our case, we wanted to approach the problem from first principles, with usability concerns driving our whole approach to the design, rather than the more common approach, which is to create a first-cut design based only on generic consideration of user concerns, that is then used as a target for more specific usability refinements based on empirical or heuristic methods. For that reason, we chose techniques that focused on the cognitive needs of programmers, rather than focusing on aspects of languages or interfaces. Attention Investment does this at a very abstract level, addressing the cognitive demands of the programming task itself. Cognitive Dimensions applies those demands in specific ways in order to anticipate usability problems. We have found the combination to be an effective approach for taking human issues into account in the design of programming languages.

7 Status of implementation and empirical work

We have implemented several "demonstration of concept" prototypes that concentrate mostly on the front end of the features we have described, i.e., the aspects the user sees. The prototypes are: (1) a set of Excel macros (written in Excel's Visual Basic for Applications) that implements many of the features of user-defined functions and matrices but for a limited example; (2) a PowerPoint mock-up that demonstrates functionality through PowerPoint's event-oriented animation features; and (3) a prototype in Macromedia Director that implements a smaller subset of the features than the Excel macro implementation but for a larger range of examples.

None of these prototypes is adequate for use in a real application or for experimental evaluation, because so much user interaction with Excel relies on seamless recalculation in response to any exploratory user action. The behind-the-scenes aspects — that is, the proper evaluation semantics of sheet instantiation and recalculation — can use many of the methods that have been fully implemented in Forms/3, which can thus be regarded as a standalone prototype for those aspects. The Forms/3 implementation has been the setting for numerous empirical studies with human subjects (c.f. [BAD+01, BCACA02], but Forms/3's front-end aspects differ from those presented here, so the Forms/3 empirical work is only a precursor of empirical studies that would evaluate the approach described in this paper.

8 Conclusion

We have presented a design process in which we started from a popular, but limited, end-user programming paradigm (the spreadsheet), and extended it to provide some of the capabilities of general purpose programming languages (user defined functions and matrices). We have not approached our work as a generic language design exercise. For example, we have explicitly chosen not to support some standard aspects of the functional paradigm, such as recursion. Instead, we have made design tradeoffs that give first priority to the cognitive requirements of the spreadsheet users.

The analytic approach that we have taken in this design process is based on recent developments in the study of programming usability, including the cognitive dimensions of notations framework, and the attention investment model of abstraction use. Both provide a systematic description of design criteria that are intuitively familiar, but usually applied only in an ad-hoc way by programming language designers. We have used these criteria to establish and review priorities in our design choices. We believe that this approach is applicable not only to "end-user" programmers (who are generally likely to have more difficulty learning and using new programming languages), but also to the design and extension of programming languages and environments for professional programmers.

This project is unusual as a programming language research project: not only because of the extent to which it highlights user concerns, but also in that it applies programming language insights to a product not normally considered as a programming language. The exercise is also distinctive with regard to other research in programming language usability, because we are working to *evolve* the design of a *well-established* language in practical use, rather than starting from scratch with the design of a new language for research purposes.

We believe putting human concerns at the forefront of language design will become increasingly important. The ability to integrate programming language principles with human problem solving principles when evolving established programming systems may in the future be the factor that differentiates successful applied programming language design research and practice.

9 References

[Amb87] A Ambler. Forms: Expanding the visualness of sheet languages. In *Workshop on Visual Languages, Linkoping, Sweden*, August 1987.

[BA94] M Burnett and A Ambler. Interactive visual data abstraction in a declarative visual programming language. *Journal of Visual Languages and Computing*, 5:29–60, March 1994.

[BAB+00] B Boehm, A Abts, S Brown, B Chulani, E Clark, R Horowitz, D Madachy, Reifer, and B Steece. *Software Cost Esimation with COCOMO II*. Prentice Hall PTR, Upper Saddle River, NJ, 2000.

[BAD+01] Margaret Burnett, John Atwood, Rebecca Walpole Djang, Herkimer Gottfried, James Reichwein, and Sherry Yang. Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. *Journal of Functional Programming*, 11:155–206, March 2001.

[BB02] AF Blackwell and M Burnett. Applying attention investment to end-user programming. In HCC'02 [HCC02], pages 28–30.

[BBC02] L Beckwith, M Burnett, and C Cook. Reasoning about many-to-many requirement relationships in spreadsheets. In HCC'02 [HCC02], pages 149–157.

[BCACA02] M Burnett, N Cao, M Arredondo-Castro, and J Atwood. End-user programming of time as an 'ordinary' dimension in grid-oriented visual programming languages. *Journal of Visual Languages and Computing*, 13(4):421–447, August 2002.

[BG87] P. Brown and J. Gould. Experimental study of people creating spreadsheets. *ACM Transactions on Office Information Systems*, 5:258–272, 1987.

[BG03] AF Blackwell and TRG Green. Notational systems – the cognitive dimensions of notations framework. In JM Carroll, editor, *HCI Models, Theories, and Frameworks: Toward an Interdisciplinary Science*. Morgan Kaufmann, 2003.

[Bla01] A Blackwell. See what you need: helping end users to build abstractions. *Journal of Visual Languages and Computing*, 5:475–499, October 2001.

[Bla02] AF Blackwell. First steps in programming: A rationale for attention investment models. In HCC'02 [HCC02], pages 2–10.

[CRB+98] E Chi, J Riedl, P Barry, P Konstan, and J Konstan. Principles for information visualization spreadsheets. *IEEE Computer Graphics and Applications*, pages 30–38, July 1998.

[CW00] C. Corritore and S. Wiedenbeck. Direction and scope of comprehension-related activities by procedural and object-oriented programmers: An empirical study. In *International Workshop on Program Comprehension*, pages 139–148, Limerick, Ireland, June 2000.

[Dav96] JS Davis. Tools for spreadsheet auditing. *International Journal of Human-Computer Studies*, 45:429–442, 1996.

[DP95] Joseph Dumas and Paige Parsons. Discovering the

way programmers think about new programming environments. *Communications of the ACM*, 38:45–56, June 1995.

[FFF+97] R Findler, C Flanagan, M Flatt, S Krishnamurthi, and M Felleisen. DrScheme: A Pedagogic Programming Environment for Scheme. In PLILP'97 [PLI97], pages 369–388.

[GP96] TRG Green and M Petre. Usability analysis of visual programming environments: a "cognitive dimensions" framework. *Journal of Visual Languages and Computing*, 7:131–174, 1996.

[GPB91] T. Green, M. Petre, and R. Bellamy. Comprehensibility of visual and textual programs: A test of superlativism against the 'match-mismatch' conjecture. In *Empirical Studies of Programmers: Fourth Workshop, New Brunswick, New Jersey*, pages 121–146. Ablex, December 1991.

[HCC02] *IEEE Conference on Human-Centric Computing Languages and Environments, Arlington*. IEEE Computer Society, September 2002.

[HLMC96] Christopher M. Hoadley, Marcia C. Linn, Lydia M. Mann, and Michael J. Clancy. When, why and how do novice programmers reuse code? In *Empirical Studies of Programmers: Sixth Workshop*. Ablex, 1996.

[HM90] C Hughes and J Moshell. Action Graphics: A spreadsheet-based language for animated simulation. In T Ichikawa, E Jungert, and R Korfhage, editors, *Visual Languages and Applications*, pages 203–235. Plenum Publishing, 1990.

[HO85] CV Hall and JT O'Donnell. Debugging in a side-effect-free programming environment. In *Proc ACM Symposium on Language Issues and Programming Environments*. ACM, Seattle, January 1985.

[Hud94] S Hudson. User interface specification using an enhanced spreadsheet model. *ACM Transactions on Graphics*, 13:209–239, July 1994.

[IMCZ98] Takeo Igarashi, Jock Mackinlay, Bay-Wei Chang, and Polle Zellweger. Fluid visualization of spreadsheet structures. In *IEEE Symposium on Visual Languages, Halifax, Nova Scotia*, pages 118–125. IEEE, September 1998.

[Lev94] M Levoy. Spreadsheet for images. *Computer Graphics*, 28:139–146, 1994.

[LGAH02] Y. Li, J. Grundy, R. Amor, and J. Hosking. A data mapping specification environment using a concrete business form-based metaphor. In HCC'02 [HCC02], pages 158–166.

[Lie84] H Lieberman. Steps toward better debugging tools for LISP. In *ACM Symposium on Lisp and Functional Programming (LFP'84)*, pages 247–255. ACM, 1984.

[Mye91] B Myers. Graphical techniques in a spreadsheet for specifying user interfaces. In *ACM Conference on Human Factors in Computing Systems, New Orleans*, pages 243–249, April 1991.

[Nar93] B. Nardi. *A Small Matter of Programming: Perspectives on End User Computing*. The MIT Press, Cambridge, MA, 1993.

[NC85] A Newell and SK Card. The prospects for psychological science in human-computer interaction. *Human-Computer Interaction*, 1:209–242, 1985.

[Nil98] Henrik Nilsson. *Declarative Debugging for Lazy Functional Languages*. PhD thesis, Department of Computer and Information Science, Linköpings universitet, S-581 83, Linköping, Sweden, May 1998.

[NM90] J Nielsen and R Molich. Heuristic evaluation of user interfaces. In *Proceedings of ACM CHI'90 Conference, Seattle*, pages 249–256, April 1990.

[Pan98] RR Panko. What we know about spreadsheet errors. *Journal of End User Computing*, 10:15–21, 1998.

[PLI97] *International Symposium on Programming Languages Implementations, Logics, and Programs (PLILP'97)*, volume 1292 of *Lecture Notes in Computer Science*. Springer Verlag, September 1997.

[PM96] JF Pane and BA Myers. Usability issues in the design of novice programming systems. Technical Report CMU-CS-96-132, Carnegie Mellon University, School of Computer Science, August 1996.

[PMM02] J Pane, B Myers, and L Miller. Using HCI techniques to design a more usable programming system. In HCC'02 [HCC02], pages 198–206.

[RLDB98] G Rothermel, L Li, C DuPuis, and M Burnett. What you see is what you test. In *International Conference on Software Engineering, Kyoto*, pages 198–207, April 1998.

[SCB96] T Smedley, P Cox, and S Byrne. Expanding the utility of spreadsheets through the integration of visual programming and user interface objects. In *Advanced Visual Interfaces '96, Gubbio, Italy*, pages 148–155, May 1996.

[SR97] J Sparud and C Runciman. Tracing lazy functional computations using redex trails. In PLILP'97 [PLI97].

[TA90] AP Tolmach and AW Appel. Debugging Standard ML without reverse engineering. In *Proc ACM Conference on Lisp and Functional Programming, Nice*. ACM, June 1990.

[US87] D. Ungar and R. Smith. Self: The power of simplicity. *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 227–242, October 1987.

[YBDZ97] S. Yang, M. Burnett, E. DeKoven, and M. Zloof. Representation design benchmarks: a design-time aid for VPL navigable static representations. *Journal of Visual Languages and Computing*, 8(5/6):563–599, Oct/Dec 1997.