# On the importance of being the right size: the challenge of conducting realistic experiments[1]

Simon Peyton Jones, University of Glasgow

simonpj@dcs.glasgow.ac.uk

## 1. INTRODUCTION

As Milner eloquently argued in a submission to the Fairclough review of the Science and Engineering Research Council, Computer Science is a unique synthesis of a *scientific* and *engineering* discipline. On the one hand, we seek to abstract unifying ideas, principles and techniques. On the other, the production of artefacts – software and/or hardware systems – is part of the very soul of the subject. Both sides are necessary. Without the ideas and principles, each artefact is constructed *ad hoc*, without taking advantage of lessons and ideas abstracted from earlier successes and failures. Without artefacts, it is impossible to know whether the ideas and principles are any use.

Unless we are aware of the symbiotic relationship between science and engineering, there is a danger that we will unconsciously construct the equations

science = research

engineering = development

No research engineer would make such an identification, because engineers realise that there are substantial research challenges in building artefacts, but computer scientists (sic) are prone to do so. Perhaps one reason for this is that computer programs, unlike bridges or electric motors, are built from a particularly malleable medium. Unlike wood or steel, computer programs do not break merely because they have become too large, nor do they rot or rust. It is simply our own frustrating inability to deal with complexity which limits our "reach" in building computer systems. In truth, though, the challenges of dealing with complexity are at least as great as those facing engineers working with physical media.

In short, the process of building systems plays a critical, and under-valued, role in computing in general, and in Computer Science research in particular. In this paper I argue the case for valuing system-building more highly than the UK academic research community generally does.

## 2. RESEARCH IN COMPUTING

Much research in Computer Science is, quite rightly, carried out by "one man[2] and his dog[3]". Where a grant is involved, the time-scale is usually three years. This is a very cost-effective way to carry out research. The participants are highly motivated, the work is tightly focused, and little time is spent on project-management overheads.

---

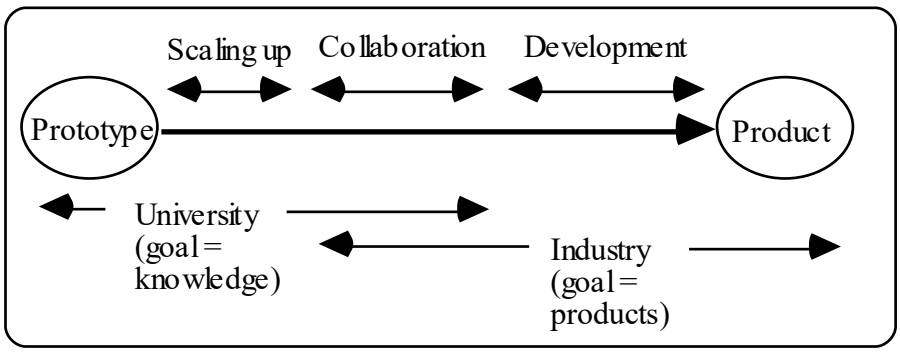
1 In "*Computing tomorrow: future research directions in computer science*", ed Wand & Milner, Cambridge University Press 1996, pp321-335

2 Or woman.

3 Or workstation.

What is interesting is what happens next, or rather what does not happen next. The results of a three-year study are generally in the form of papers, and perhaps a prototype of a promising piece of software. Sometimes the plain fact is that the results are not very encouraging, and a new approach is needed. But sometimes, especially when a particular area has received attention from several small-scale projects, the appropriate next step is to "scale up" the prototype into a larger, more robust system.

The difficulty is that such a scaling-up enterprise often falls between the two stools of academic respectability on the one hand (since it can readily be dismissed as "just development work") and industrial relevance on the other (since the prototype is often a long way from a commercial product). The following picture illustrates the dilemma:




Once a company perceives that an idea offers benefits (especially products or productivity) in less than three years or so, there is a reasonable chance that a collaborative project can be established which, if successful, will lead on to subsequent development work funded entirely by the company. The difficulty is that the gap between a prototype and a product is often much longer than three years. The result is, of course, the oft-cited "British disease" of producing lots of good ideas, but failing to take enough of them through to successful commercial products. Somewhat more support for scaling-up work might contribute to a cure.

It is reasonable to ask, then: is scaling up a task appropriate for a university research department? The answer is a resounding yes! My main thesis is this:

> The twin exercises of
>
> > (a) scaling up a research prototype to a robust, usable system, and
> >
> > (b) studying its behaviour when used in real applications,
>
> expose new *research* problems which could not otherwise be identified and studied.

Scaling up is not a routine development process. New technical challenges come to light when building a "real" system, that were either not apparent, or were relatively unimportant, in the prototype stage. Identifying and solving these problems constitutes a durable contribution to knowledge, provided of course that the knowledge is abstracted and published, rather than simply being encoded into the scaled-up prototype.

Scaling up also provides a new opportunity: that of establishing a symbiosis between the emerging robust system on the one hand, and some realistic applications on the other. These applications can give much more "bite" and focus to the scaling-up effort, and make sure that effort is being directed to where it will be of most use. This interplay often cannot be established at the prototype stage, because the prototype is too small and fragile to handle a real application.

It is not difficult to think of examples of successful scaling-up projects. The York Ada compiler, the Edinburgh LCF system, the HOL theorem prover, Wirth's Pascal and Modula compilers. A

contemporary example is the Fox project at CMU, which is aimed at implementing the complete TCP/IP protocol stack in Standard ML, to see whether the job can be done in a more modular and robust manner than the conventional technology. To illustrate the ideas outlined so far, the remainder of this paper describes another scaling-up project with which I have been closely involved.

## 3. FUNCTIONAL PROGRAMMING AND THE GLASGOW HASKELL COMPILER

I am fortunate to have been supported by a succession of SERC grants to work on the implementation of non-strict functional programming languages, on both sequential and parallel hardware. The most visible outcomes of this work are the Glasgow Haskell Compiler (`GHC`), and the GRIP multiprocessor. I believe that these are useful artefacts, but probably their most lasting value lies in the research issues their construction exposed, the collaborations they have enabled, the informal standardisation they have nourished, and the framework they have provided for the research of others. I elaborate on these in the following sections. (I use the third person throughout, since the work was done with a number of other colleagues.)

In order to makes sense of what follows, one needs to have some idea of what functional programming is, so I begin with a brief introduction.

### 3.1. Functional programming

Anyone who has used a spreadsheet has experience of functional programming. In a spreadsheet, one specifies the value of each cell in terms of the values of other cells. The focus is on *what* is to be computed, not *how* it should be computed. For example:

- we do not specify the order in which the cells should be calculated – instead we take it for granted that the spreadsheet will compute cells in an order which respects their dependencies.
- we do not tell the spreadsheet how to allocate its memory – rather, we expect it to present us with an apparently infinite plane of cells, and to allocate memory only to those cells which are actually in use.
- for the most part, we specify the value of a cell by an *expression* (whose parts can be evaluated in any order), rather by a *sequence of commands* which computes its value.

An interesting consequence of the spreadsheet's unspecified order of re-calculation is that the notion of *assignment* , so pervasive in most programming languages, is not very useful. After all, if you don't know exactly when an assignment will happen, you can't make much use of it!

Another well-known nearly-functional language is the standard database query language SQL. An SQL query is an expression involving projections, selections, joins and so forth. The query says what relation should be computed, without saying how it should be computed. Indeed, the query can be evaluated in any convenient order. SQL implementations often perform extensive query optimisation which (among other things) figures out the best order in which to evaluate the expression.

Spreadsheets and databases, then, incorporate *specialised, not-quite-functional* languages. It is interesting to ask what you get if you try to design a *general-purpose, purely-functional* language. Haskell is just such a language. To give an idea of what Haskell is like, Figure 1 gives a Haskell fuction which sorts a sequence of integers using the standard Quicksort algorithm, and an explanation of how the function works. For comparison, Figure 2 gives the same function written in C.

It is interesting to compare the two:

- The Haskell function is a great deal shorter. Once one has overcome the initial unfamiliarity, it is also much,much easier to convince oneself (formally or informally) that the Haskell function is correct than it is for the C version. Indeed, it is so easy to make a small but fatal error when weriting the C version that I copied it out of a textbook.
- On the other hand, the C version incorporates Hoare's very ingenious technique which allows the sequence to be sorted without using any extra space. In contrast, the Haskell program deliberately leaves memory allocation unspecified. As a direct result, functional programs usually use more space (sometimes much more space) than their imperative counterparts, because programmers are cleverer than compilers at optimising space usage.

Functional languages take another large step towards a higher-level programing model. Programs are easier to design, write and maintain, but the language offers the programmer less control over the program's execution. This is a familiar tradeoff. We all stopped writing assembly-language programs, except perhaps for key inner loops, long ago, because the benefits of using a high-level language (an arbitrary number of named, local variables instead of a fixed number of registers, for example) far outweigh the modest run-time costs. Similarly, we willingly accept the costs of a virtual

```
qsort []     = []
qsort (x:xs) = elts_lt_x ++ [x] ++ elts_greq_x
               where
                 elts_lt_x   = [y | y <- xs, y < x]
                 elts_greq_x = [y | y <- xs, y >= x]
```

Here is an explanation of how `qsort` works. The first line reads: "The result of sorting an empty list (written `[]`) is an empty list".

The second line reads: "To sort a list whose first element is `x` and the rest of which is called `xs`, just sort all the elements of `xs` which are less than `x` (call them `elts_lt_x`), sort all the elements of `xs` which are greater than or equal to `x` (call them `elts_greq_x`), and concatenate (`++`) the results, with `x` sandwiched in the middle."

The definition of `elts_lt_x`, which is given immediately below, is read like this: "`elts_lt_x` is the list of all `y`'s such that `y` is drawn from the list `xs`, and `y` is less than `x`". The definition of `elts_greq_x` is similar.

The syntax is deliberately reminiscent of standard mathematical set notation, pronouncing "`|`" as "such that" and "`<-`" as "drawn from".

When asked to sort a non-empty list, `qsort` calls itself to sort `elts_lt_x` and `elts_greq_x`. That's OK because both these lists are smaller than the one originally given to `qsort`, so the splitting-and-sorting process will eventually reduce to sorting an empty list, which is done rather trivially by the first line of `qsort`.

**Figure 1**: Quicksort in Haskell

```
qsort( a, lo, hi )
int a[], hi, lo;
{
  int h, l, p, t;

  if (lo < hi) {
    l = lo;
```

```
    h = hi;
    p = a[hi];

    do {
      while (l < h) & (a[i] <= p) do
          l = l+1;
      while (h > l) & (a[h] >= p) do
          h = h-1;
      if (l < h) {
          t = a[l];
          a[l] = a[h];
          a[h] = t;
      }
    } while (l < h);

    t = a[l];
    a[l] = a[hi];
    a[hi] = t;

    qsort( a, lo, l-1 );
    qsort( a, l+1, hi );
  }}
```

**Figure 2**: Quicksort in C

memory paging system, in exchange for the more supportive programming model of an infinite virtual address space; the days of explicit memory overlays are over.

The interesting question is, of course, whether the benefits of a higher level of abstraction outweigh the costs of extra run-time resource consumption. In order to even begin to answer that question, one has to build an implementation which

(a) is sophisticated enough to have reduced the run-time costs to a minimum;

(b) is robust enough that real programmers can use it for real problems.

The Glasgow Haskell compiler aspires to being such an implementation.

## 3.2. Standardisation

While one is building throwaway prototypes, there is little or no incentive to contribute to standardisation. On the other hand, when one contemplates investing substantial effort in a (hopefully) long-lived system, the picture changes entirely. Without some sort of standard interface (a programming language in our case) nobody will use the system, or want to invest their own efforts in developing it further.

Until about 1990 there was a fairly anarchic situation in the non-strict functional programming community, in which each research group had its own language, and no common standard had emerged. (This contrasted with the strict functional language community which had adopted Standard ML by this time.) Our interest in scaling up led us to become involved in an international effort to agree a common language, which successfully developed the Haskell language. Many others participated in this process, but most were motivated by a desire to get non-strict functional programming into a state where it had some hope of escaping from the computer lab and into the real world – in short, scaling up. Haskell is not a formal standard, but now that many research groups around the world are using Haskell, it has certainly become a *de facto* standard.

## 3.3. Benchmarking

Nowadays, computer systems usually come rated with their "SPEC-marks". This relates to the speed at which they can execute the SPEC benchmark suite. Unlike earlier synthetic benchmarks (such as Whetstone), SPEC consists of a collection of real application programs, across a variety of application domains. Small programs with dominant inner loops, which are susceptible to special pleading in an implementation, have been weeded out. As a direct result, more of designers' creativity has been focused on solving the problems which matter, rather than merely on making Whetstone go fast.

Our focus on practical system-building has led us to attempt, in a modest way, to achieve the same goals for the non-strict functional programming community. We have collected a variety of large, real application programs, written by people other than ourselves who were simply trying to get a job done (not benchmark a compiler). The resulting suite, the `nofib` suite, is now directly available to others.

We use the `nofib` suite on a daily basis as a stress test for our compilation technology. Doing so keeps us honest: deficiencies in our compiler quickly become painfully obvious. A second important use for the suite is as a basis for making detailed quantitative measurements of the behaviour of "real" functional programs.

The fact that these programs were written in the first place was due, in turn, to the existence of a standard language (Haskell in this case), and to the existence of implementations of Haskell which were robust enough that programmers could treat them as a tool rather than as an object of study.

## 3.4. Application

Some four years ago we set up the FLARE project, whose goal was to study the effectiveness (or otherwise) of functional programming in real applications, using the compilers and other tools produced by the Glasgow team. The FLARE participants were deliberately not functional programmers; rather, they were application-domain experts who were prepared to try their hand at functional programming. The applications included a theorem prover, numerical hydrodynamics, data-compression utilities and graphical user interfaces.

In retrospect, we were much too ambitious. From our point of view as implementors, FLARE was an enormous success, serving as a major source of direction and focus for our work. But our tools and implementations were, until the final stages of FLARE, embarrassingly immature, especially those for our parallel implementation on the GRIP multiprocessor. As a result our poor application partners had a frustrating time of it.

The lessons we would draw from this experience are these:

- Using a collection of applications as a "test load", to guide (or even govern) priorities, is extremely beneficial.
- The application partners must be aware from the start that they are participating in an experiment, not in a product development. Even then, they need a saintly disposition.

## 3.5. Research benefits

Earlier, I mentioned that a most important outcome of the scaling-up exercise is the research payoff, in the form of research issues identified and solved. Our Haskell compiler illustrates the point well. Here are a number of examples:

- **Input/output.** Purely functional languages have usually provided input/output mechanisms which are rather inconvenient to use, because input/output is inherently a side-effecting activity, rather alien to an otherwise purely-functional setting. As we distributed our compiler more widely, it became clear that difficulties with I/O were a major obstacle to many potential users.

  This painful realisation led us to focus our attention on the I/O problem. Serendipitously, this coincided with work done by one of our colleagues (Wadler) on so-called monads. We discovered how to apply monads to allow I/O-performing computations to be expressed much more easily than before, and to be implemented rather efficiently.

  A neat generalisation allowed direct calls from Haskell to C, a mechanism on which we built the whole of the new I/O system in `GHC`. Based on the demands of our users, we are now developing the ideas further, to include interrupts, timeouts, and call-backs.

- **Encapsulated state.** An obvious variation of an I/O-performing computation is a computation which performs some side effects on state which is entirely internal to a program. Some (though not many) algorithms appear to be expressible only using mutable state – depth-first search, for example.

  Driven by this observation, we have recently succeeded in generalising the I/O-monad idea to accommodate securely-encapsulated computations which manipulate internal mutable state in the program, while presenting a purely-functional external interface to the rest of the program.

- **Profiling**. We have always known that functional programs (especially non-strict ones) sometimes consume much more space and/or time than expected. But the advent of real applications turned this issue from a general awareness into a pressing problem.

  Measuring where the time is spent in a non-strict, higher-order language is quite tricky. For example, if one part of a program produces a list which is consumed by another part, the list is only produced as consumer demands it. This means that execution alternates between producer and consumer, so it becomes much harder to measure how much time is spent in each part of the program.

  The existence of higher-order functions makes things worse: in a higher-order program it is hard to say just what a "part" of a program, to which execution costs should be attributed, means any more.

  We have succeeded in developing a profiler which solves these problems, the first to provide accurate time profiling for a compiled, non-strict, higher-order language. Its implementation in the `GHC` provides a convincing demonstration that the ideas work, and do so with an acceptably low overhead. Indeed, many developments of the profiler followed directly from experience of its use.

- **Unboxed data types**. In a non-strict language, a function which takes an integer argument will usually be passed a heap-allocated *suspension* which, when evaluated, produces the integer. To support this argument-passing convention, all integers, whether evaluated or not, are represented by a heap-allocated object or "box". In order to do any operations on an integer, it must first be evaluated, extracted from its box, operated on, and then boxed again.

  These boxing and unboxing operations are usually implicit, and handled by the code generator. Unfortunately, that forces into the code generator an important class of optimisations, which aim to manipulate a value in unboxed form for as long as possible. For

example, there is little point in adding one to an integer, boxing the result, and then unboxing it again before performing some subsequent operation on it. Getting the code generator to do a really good job of avoiding redundant operations is possible, but it is not easy.

We have developed an alternative approach, in which unboxed values become explicitly part of the language, with a well-defined semantics. The boxing and unboxing operations thereby also become explicit, and a set of generally-useful program transformations can be used to eliminate redundant operations. The desire for a tidier, more modular implementation led us to develop a useful new theory, whose applicability goes well beyond our own compiler.

This catalogue is not intended as a demonstration of research prowess. The point is simply that most of these issues are either unimportant or inaccessible in a small prototype implementation, but become pressing matters when writing and compiling large Haskell programs. The exercise of scaling up remorselessly exposed new challenges. (The glory of working in a university is, of course, that every new problem can become the object of a new research project, rather than simply being an obstacle.)

## 3.6. The compiler as motherboard

There is a second major crop of research payoffs from the scaling-up exercise. Often individual researchers in functional-language implementations have an idea, perhaps a good idea, which they want to try out. For example, they might have devised a new way of analysing programs which should improve the implementation, or a new technique for code generation, or a new program-transformation. The difficulty is that to try it out they have to build a great deal of scaffolding: often this takes the form of a complete implementation of a very small functional language, complete with front end, symbol table, code generation, and so on. It has to be a very small language in order to make the scaffolding feasible at all, and even then it is quite an effort.

The outcome of such work is often extremely unconvincing. Even if the idea gives good results on the half-dozen tiny programs on which its behaviour is measured, what does that say about its behaviour when given a 30,000 line program? A major goal of our compiler was to resolve this dilemma by providing a well-structured "motherboard" into which researchers can "plug in" their incremental improvements. The front end, back end, and (most important) suite of benchmark programs, are all provided. The effort of integrating one's ideas into a large existing compiler is still not small, but it is much smaller than that of making a complete new implementation for even a small language, and the results are incomparably more convincing. Not only that, but if the project is successful, its results can be used immediately by others.

Here are some examples of the ways in which our compiler has already been used as a "motherboard", mostly by PhD students:

- **Generational garbage collection** has been used for languages such as Lisp and Standard ML for some while. The folklore was that it would work badly for *non-strict* languages such as Haskell, because suspensions are continually being updated, an operation which is relatively expensive for generational collectors.

  Based on measurements of substantial Haskell programs we have been able to show that, on the contrary, generational collectors work very well for non-strict programs, because objects are almost always updated when they are very young. The compiler we distribute now has a generational collector as standard.

- **Deforestation**. Functional programmers tend to make heavy use of lists (and other data structures) as intermediate values connecting a pipeline of computations together. For example, consider the expression

```
map f (map g xs)
```

The function `map` applies a function (its first argument) to each element of a list (its second argument). The expression given above therefore first applies `g` to each element of `xs`, builds a list of those results, and applies f to each element of this list. An intermediate list is built, only to be consumed immediately. The program is easier to understand in this form, but less efficient to execute than one in which the computations are entwined together with no intermediate structures. For example, here is an equivalent expression:

```
map (f.g) xs
```

A single map suffices to apply the function `f`-composed-with-`g`, `(f.g)`, to the list `xs`. It is well known that the intermediate lists can often be transformed out, a process known as deforestation, but the process has so far been too complex, or incomplete, to automate. We have recently developed two new approaches to deforestation which can be completely automated, and are incorporating them in the compiler. As in other cases, the motivation was provided by the need for practical, rather than idealised, solutions. Our experience with implementing our deforestation techniques in `GHC` led directly to new developments.

- **Strictness analysis** is a static program analysis which aims to discover when a given function is sure to evaluate its argument(s), and to what degree. Using this information it is often possible to derive a more efficient calling convention for the function. A very large number of papers have been written about strictness analysis, but only a tiny minority report the effectiveness of the analysis on any but tiny examples. The reason is exactly that outlined above: they lack adequate scaffolding.

  We have used our compiler to make detailed measurements of a fairly simple strictness analyser, and we know of others who are building more sophisticated analysers for the compiler.

- **Compilation by transformation**. A unifying theme of the compiler is the use of program transformation as the major compilation technique. The program being compiled is translated into a simple "Core" language, and is then extensively transformed, before being fed to the code generator. This is not, of course, an original idea, but having the transformation system embedded in a substantial compiler has enabled a PhD student to make extensive quantitative measurements of the effective of a variety of transformation strategies. These, in turn, led him to propose, implement, and measure new transformations which we had not at first thought of.

## 3.7. Research vs development

A major tension in this kind of project is between the research goals on the one hand, and the care and maintenance of the artefact itself on the other. It is all too easy for the compiler to take on a life of its own, and to absorb all the limited effort we have available. We come under pressure from our users to enhance it in one way or another, and it is, in any case, all too easy to become addicted to adding "one more feature". One can address each such demand in one of three ways:

- **Resist it**, on the grounds that it isn't research, involves too much work, or that it opens up too big a research area. Examples of desirable developments we have not undertaken for

these reasons are: an interactive version of our compiler, a persistent store, and a `dynamic` data type to allow run-time type checking where required.

- **Try to satisfy it as directly and economically as possible**, because it will make a real impact on the usefulness and acceptability of the system. Requests that fall into this category include porting to other machine architectures, improving compilation speed, reducing compiled code size, and allowing Haskell to manipulate pointers into `malloc`'d C space.
- **Treat it as a new research opportunity**. We took this approach with input/output, mixed-language working, profiling, and graphical user interfaces.

The important thing is to recognise and embrace the third option: a major justification for the whole scaling-up exercise is, after all, to expose new research problems.

## 4. CONCLUSIONS

I have argued that the exercise of scaling up a research prototype into a robust implementation, if carefully undertaken, is a legitimate, fruitful and important form of research. When it goes well it can set up a virtuous circle of benefits:

- it exposes new research challenges which could not otherwise be identified and studied;
- it makes available a useful tool for others to use, which may in turn open up new areas of application for the technology, leading to new challenges;
- it provides valuable scaffolding to support the research of others in the same area;
- it contributes to standardisation;
- it allows credible quantitative measurements to be made;
- it bridges an important part of the gap between a proof-of-concept prototype and a commercial product.

So far as funding policy is concerned, I would argue that those who fund research should be willing to support some well-thought-out scaling-up projects. They are likely to be more expensive than the more common "develop-an-idea" project, and should be rigorously scrutinised to make sure that the objectives concern research rather than artefacts. The whole argument is really directed more towards modifying the cultural assumptions of referees than towards changing any official funding policy.

Building things is tremendously exciting; and much research lies in the building, as well as in the original vision.