How to write a great research paper

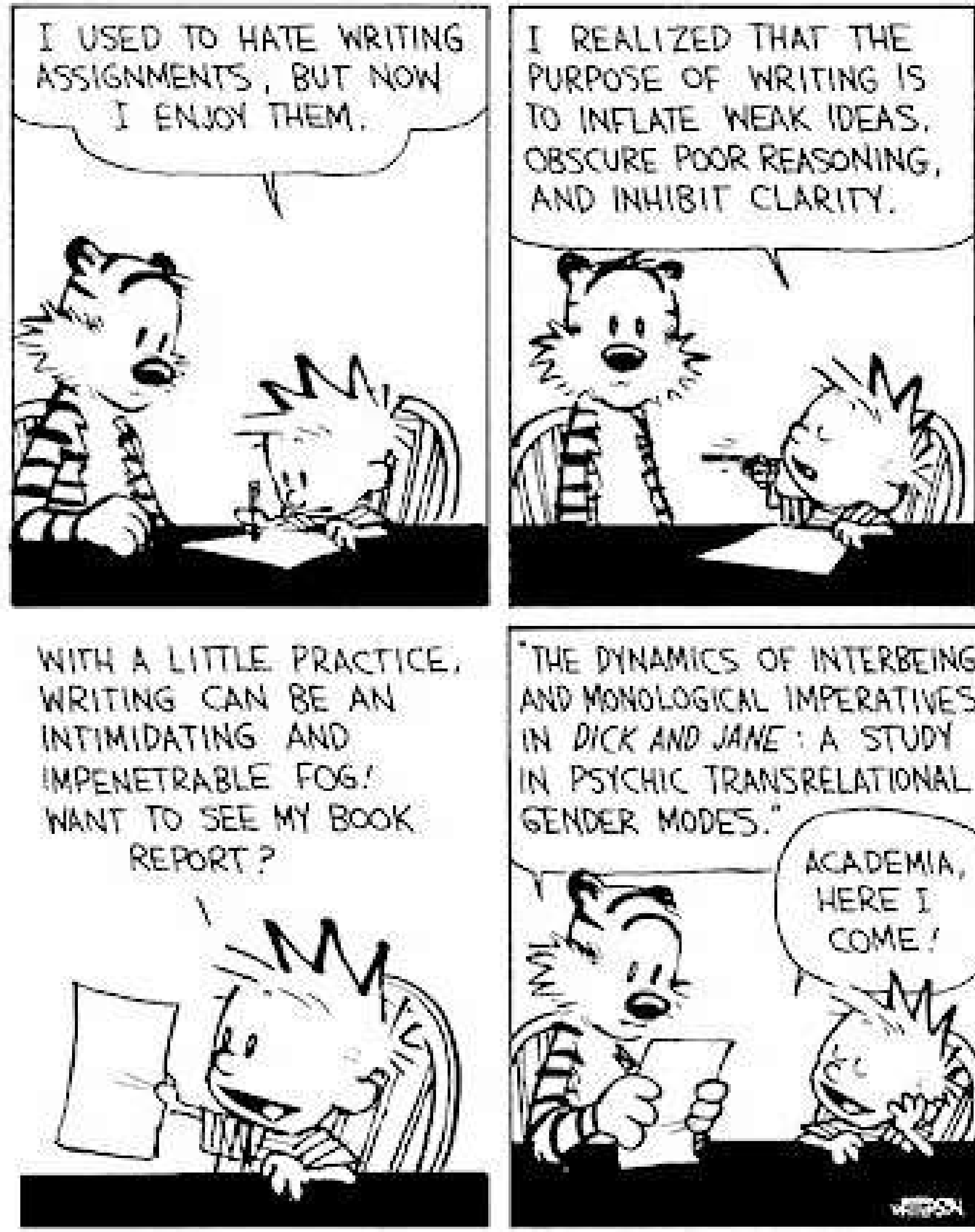
Simon Peyton Jones

Microsoft Research, Cambridge

Why bother?

Fallacy

we write papers and give talks mainly to impress others, gain recognition, and get promoted

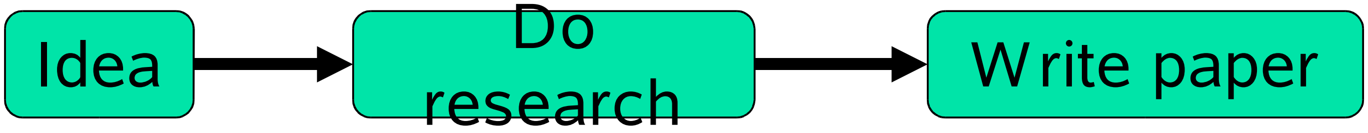


Papers communicate ideas

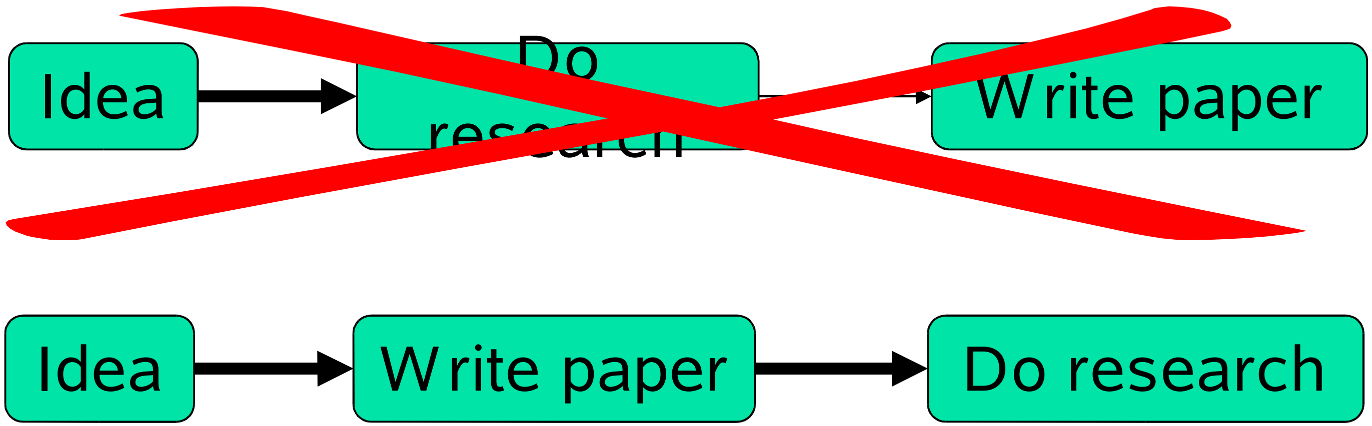
- Your goal: to infect the mind of your reader with **your idea**, like a virus
- Papers are far more durable than programs (think Mozart)

The greatest ideas are (literally) worthless if you keep them to yourself

Writing papers: model 1



Writing papers: model 2



- Forces us to be clear, focused
- Crystallises what we don't understand
- Opens the way to dialogue with others: reality check, critique, and collaboration

Do not be intimidated

Fallacy You need to have a fantastic idea before you can write a paper or give a talk. (Everyone else seems to.)

Write a paper,
and give a talk, about
any idea,
no matter how weedy and insignificant it may seem to you

Do not be intimidated

Write a paper, and give a talk, about any idea, no matter how insignificant it may seem to you

- **Writing the paper is how you develop the idea in the first place**
- It usually turns out to be more interesting and challenging that it seemed at first

The purpose of your paper

The purpose of your paper is...

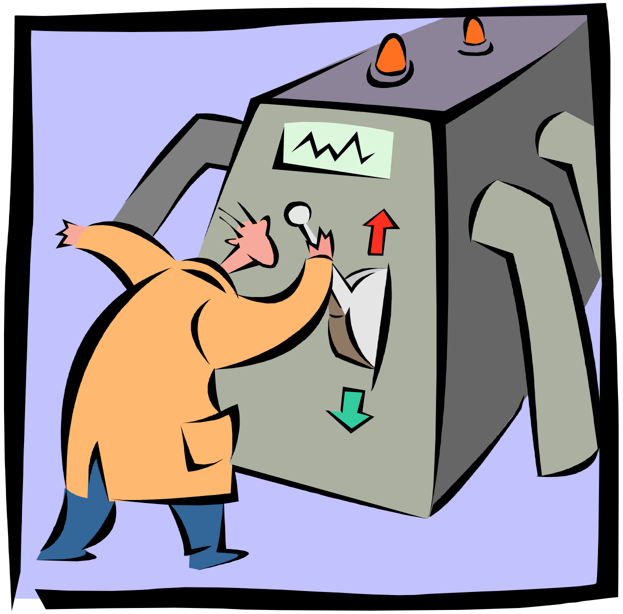
To convey your idea

...from your head to your reader's head

Everything serves this single goal

The purpose of your paper is not...

To describe the WizWoz system



- Your reader does not have a WizWoz
- She is primarily interested in re-usable brain-stuff, not executable artefacts

Conveying the idea

I wish I knew how to solve that!

- Here is a problem
- It's an interesting problem
- It's an unsolved problem
- **Here is my idea**
- My idea works (details, data)
- Here's how my idea compares to other people's approaches

I see how that works. Ingenious!

Structure

- Abstract (4 sentences)
- Introduction (1 page)
- The problem (1 page)
- My idea (2 pages)
- The details (5 pages)
- Related work (1-2 pages)
- Conclusions and further work (0.5 pages)

The abstract

- I usually write the abstract last
- Used by program committee members to decide which papers to read
- Four sentences [Kent Beck]
    1. State the problem
    2. Say why it's an interesting problem
    3. Say what your solution achieves
    4. Say what follows from your solution

Example

1. Many papers are badly written and hard to understand
2. This is a pity, because their good ideas may go unappreciated
3. Following simple guidelines can dramatically improve the quality of your papers
4. Your work will be used more, and the feedback you get from others will in turn improve your research

Structure

- Abstract (4 sentences)
- **Introduction** (1 page)
- The problem (1 page)
- My idea (2 pages)
- The details (5 pages)
- Related work (1-2 pages)
- Conclusions and further work (0.5 pages)

The introduction (1 page)

1. Describe the problem
2. State your contributions

...and that is all

Describe the problem

1 Introduction

There are two basic ways to implement function application in a higher-order language, when the function is unknown: the *push/enter* model or the *eval/apply* model [11]. To illustrate the difference, consider the higher-order function **zipWith**, which zips together two lists, using a function **k** to combine corresponding list elements:

```
zipWith :: (a->b->c) -> [a] -> [b] -> [c]
zipWith k []     []     = []
zipWith k (x:xs) (y:ys) = k x y : zipWith xs ys
```

Here **k** is an *unknown function*, passed as an argument; global flow analysis aside, the compiler does not know what function **k** is bound to. How should the compiler deal with the call **k x y** in the body of **zipWith**? It can't blithely apply **k** to two arguments, because **k** might in reality take just one argument and compute for a while before returning a function that consumes the next argument; or **k** might take three arguments, so that the result of the **zipWith** is a list of functions.

Use an example to introduce the problem

State your contributions

- Write the list of contributions first
- The list of contributions drives the entire paper: the paper *substantiates* the claims you have made
- Reader thinks "gosh, if they can really deliver this, that's be exciting; I'd better read on"

State your contributions

Which of the two is best in practice? The trouble is that the evaluation model has a pervasive effect on the implementation, so it is too much work to implement both and pick the best. Historically, compilers for strict languages (using call-by-value) have tended to use eval/apply, while those for lazy languages (using call-by-need) have often used push/enter, but this is 90% historical accident — either approach will work in both settings. In practice, implementors choose one of the two approaches based on a qualitative assessment of the trade-offs. In this paper we put the choice on a firmer basis:

- We explain precisely what the two models are, in a common notational framework (Section 4). Surprisingly, this has not been done before.
- The choice of evaluation model affects many other design choices in subtle but pervasive ways. We identify and discuss these effects in Sections 5 and 6, and contrast them in Section 7. There are lots of nitty-gritty details here, for which we make no apology — they were far from obvious to us, and articulating these details is one of our main contributions.

  In terms of its impact on compiler and run-time system complexity, eval/apply seems decisively superior, principally because push/enter requires a stack like no other: stack-walking

Bulleted list of contributions

Do not leave the reader to guess what your contributions are!

Contributions should be refutable

We describe the WizWoz system. It is really cool.	We give the syntax and semantics of a language that supports concurrent processes (Section 3). Its innovative features are...
We study its properties	We prove that the type system is sound, and that type checking is decidable (Section 4)
We have used WizWoz in practice	We have built a GUI toolkit in WizWoz, and used it to implement a text editor (Section 5). The result is half the length of the Java version.

No "rest of this paper is..."

- Not: "The rest of this paper is structured as follows. Section 2 introduces the problem. Section 3 ... Finally, Section 8 concludes".

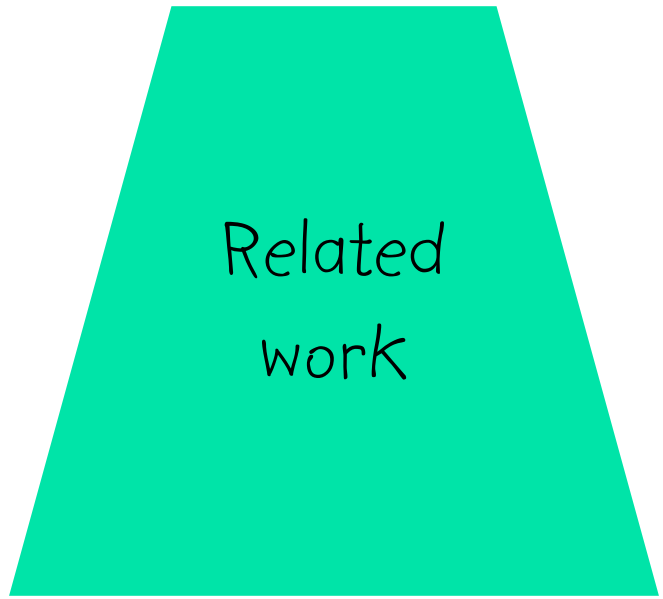
- Instead, **use forward references from the narrative in the introduction**.
  The introduction (including the contributions) should survey the whole paper, and therefore forward reference every important part.

Structure

- Abstract (4 sentences)
- Introduction (1 page)
- The problem (1 page)
- My idea (2 pages)
- The details (5 pages)
- Related work (1-2 pages)
- Conclusions and further work (0.5 pages)

No related work yet!

Your reader

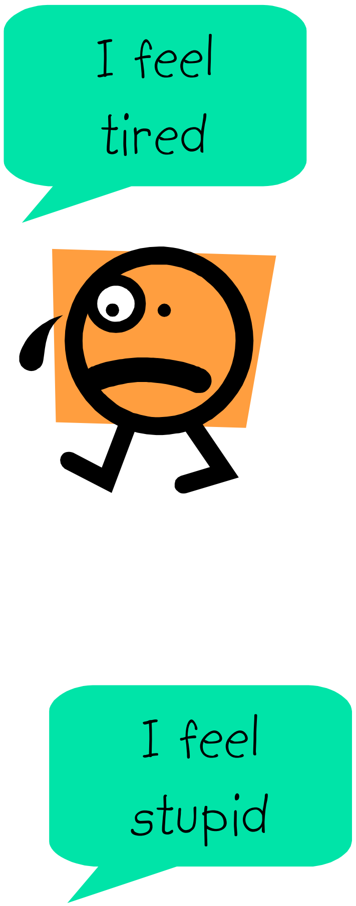


Your idea

We adopt the notion of transaction from Brown [1], as modified for distributed systems by White [2], using the four-phase interpolation algorithm of Green [3]. Our work differs from White in our advanced revocation protocol, which deals with the case of priority inversion as described by Yellow [4].

No related work yet

- Problem 1: describing alternative approaches gets between the reader and your idea
- Problem 2: the reader knows nothing about the problem yet; so your (carefully trimmed) description of various technical tradeoffs is absolutely incomprehensible



Instead...

Concentrate single-mindedly on a narrative that

- Describes the problem, and why it is interesting
- Describes your idea
- Defends your idea, showing how it solves the problem, and filling out the details

On the way, cite relevant work in passing, but defer discussion to the end

The payload of your paper

Consider a bufircuated semi-lattice D, over a hyper-modulated signature S. Suppose pi is an element of D. Then we know for every such p_i there is an epi-modulus j, such that $p_j < p_i$.

- Sounds impressive...but
- Sends readers to sleep
- In a paper you MUST provide the details, but FIRST convey the idea

The payload of your paper

Introduce the problem, and your idea, using

EXAMPLES

and only then present the general case

Using examples

The Simon PJ question: is there any typewriter font?

2 Background

To set the scene for this paper, we begin with a brief overview of the *Scrap your boilerplate* approach to generic programming. Suppose that we want to write a function that computes the size of an arbitrary data structure. The basic algorithm is "for each node, add the sizes of the children, and add 1 for the node itself". Here is the entire code for `gsize`:

```
gsize :: Data a => a -> Int
gsize t = 1 + sum (gmapQ gsize t)
```

The type for `gsize` says that it works over any type a, provided a is a *data* type — that is, that it is an instance of the class `Data`[1]. The definition of `gsize` refers to the operation `gmapQ`, which is a method of the `Data` class:

```
class Typeable a => Data a where
  ...other methods of class Data...
  gmapQ :: (forall b. Data b => b -> r) -> a -> [r]
```

Example right away

Conveying the idea

- Explain it as if you were speaking to someone using a whiteboard
- Conveying the intuition is primary, not secondary
- Once your reader has the intuition, she can follow the details (but not vice versa)
- Even if she skips the details, she still takes away something valuable

Evidence

- Your introduction makes claims
- The body of the paper provides **evidence to support each claim**
- Check each claim in the introduction, identify the evidence, and forward-reference it from the claim
- Evidence can be: analysis and comparison, theorems, measurements, case studies

Structure

- Abstract (4 sentences)
- Introduction (1 page)
- The problem (1 page)
- My idea (2 pages)
- The details (5 pages)
- **Related work** (1-2 pages)
- Conclusions and further work (0.5 pages)

Related work

Fallacy To make my work look good, I have to make other people's work look bad

The truth: credit is not like money

Giving credit to others does not diminish the credit you get from your paper

- Warmly acknowledge people who have helped you
- Be generous to the competition. "In his inspiring paper [Foo98] Foogle shows.... We develop his foundation in the following ways..."
- Acknowledge weaknesses in your approach

Credit is not like money

Failing to give credit to others can kill your paper

If you imply that an idea is yours, and the referee knows it is not, then either

- You don't know that it's an old idea (bad)
- You do know, but are pretending it's yours (very bad)

Making sure related work is accurate

- A good plan: when you think you are done, send the draft to the competition saying "could you help me ensure that I describe your work fairly?".
- Often they will respond with helpful critique
- They are likely to be your referees anyway, so getting their comments up front is jolly good.

The process

- Start early. Very early.
  - Hastily-written papers get rejected.
  - Papers are like wine: they need time to mature
- Collaborate
- Use CVS to support collaboration

Getting help

Get your paper read by as many friendly guinea pigs as possible

- Experts are good
- Non-experts are also very good
- Each reader can only read your paper for the first time once! So use them carefully
- Explain carefully what you want ("I got lost here" is much more important than "wibble is mis-spelt".)

Listening to your reviewers

Every review is gold dust

Be (truly) grateful for criticism as well as praise

This is really, really, really hard

But it's really, really, really, really, really, really important

Listening to your reviewers

- Read every criticism as a positive suggestion for something you could explain more clearly
- DO NOT respond "you stupid person, I meant X". Fix the paper so that X is apparent even to the stupidest reader.
- Thank them warmly. They have given up their time for you.

Language and style

Basic stuff

- Submit by the deadline
- Keep to the length restrictions
  - Do not narrow the margins
  - Do not use 6pt font
  - On occasion, supply supporting evidence (e.g. experimental data, or a written-out proof) in an appendix
- Always use a spell checker

Visual structure

- Give strong visual structure to your paper using
  - sections and sub-sections
  - bullets
  - *italics*
  - laid-out code
- Find out how to draw pictures, and use them

Visual structure

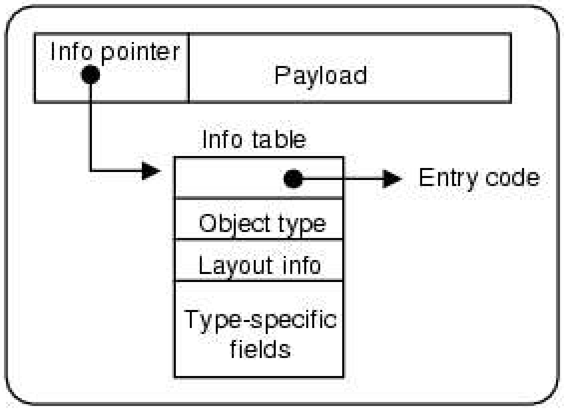


Figure 3. A heap object

The three cases above do not exhaust the possible forms of f. It might also be a $THUNK$, but we have already dealt with that case (rule THUNK). It might be a CON, in which case there cannot be any pending arguments on the stack, and rules UPDATE or RET apply.

4.3 The eval/apply model

The last block of Figure 2 shows how the eval/apply model deals with function application. The first three rules all deal with the case of a FUN applied to some arguments:

- If there are exactly the right number of arguments, we behave exactly like rule KNOWNCALL, by tail-calling the function. Rule EXACT is still necessary — and indeed has a direct counterpart in the implementation — because the function might not be statically known.
- If there are too many arguments, rule CALLK pushes a *call*

remainder of the object is called the *payload*, and may consist of a mixture of pointers and non-pointers. For example, the object $CON(C\ a_1 \ldots a_n)$ would be represented by an object whose info pointer represented the constructor C and whose payload is the arguments $a_1 \ldots a_n$.

The info table contains:

- Executable code for the object. For example, a FUN object has code for the function body.
- An object-type field, which distinguishes the various kinds of objects (FUN, PAP, CON etc) from each other.
- Layout information for garbage collection purposes, which describes the size and layout of the payload. By "layout" we mean which fields contain pointers and which contain non-pointers, information that is essential for accurate garbage collection.
- Type-specific information, which varies depending on the object type. For example, a FUN object contains its arity; a CON object contains its constructor tag, a small integer that distinguishes the different constructors of a data type; and so on.

In the case of a PAP, the size of the object is not fixed by its info table; instead, its size is stored in the object itself. The layout of its fields (e.g. which are pointers) is described by the (initial segment of) an argument-descriptor field in the info table of the FUN object which is always the first field of a PAP. The other kinds of heap object all have a size that is statically fixed by their info table.

A very common operation is to jump to the entry code for the object, so GHC uses a slightly-optimised version of the representation in Figure 3. GHC places the info table at the addresses *immediately*

Use the active voice

The passive voice is "respectable" but it DEADENS your paper. Avoid it at all costs.

NO	YES
It can *be seen* that...	We can *see* that...
34 tests were run	We ran 34 tests
These properties were thought desirable	We wanted to retain these properties
It might *be* thought that this would *be* a type error	You might think this would *be* a type error

"We" = you and the reader

"We" = the authors

"You" = the reader

Use simple, direct language

NO	YES
The object under study was displaced horizontally	The ball moved sideways
On an annual basis	Yearly
Endeavour to ascertain	Find out
It could be considered that the speed of storage reclamation left something to be desired	The garbage collector was really slow

Summary

If you remember nothing else:

- Identify your key idea
- Make your contributions explicit
- Use examples

A good starting point:

"Advice on Research and Writing"

http://www-2.cs.cmu.edu/afs/cs.cmu.edu/user/mleone/web/how-to.html