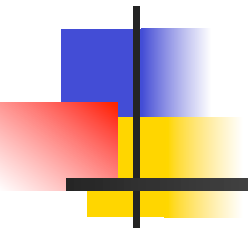


How to write a good report

Tony Field

Based on Simon Peyton Jones's
"Research Skills"
(
http://research.microsoft.com/~simonpj/papers/giving-a-talk/giving-a-talk.htm)

Use LaTeX!

Interim Report/Review Deadlines

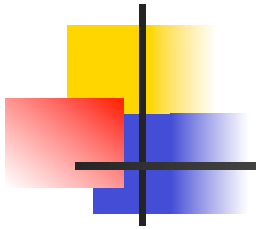
- Friday 31st January: MEng/JMC4 report due
- Friday 21st February: BEng/JMC3 report due

- Thursday 20th February: MEng and JMC4 review deadline
- Friday 7th March: BEng/JMC3 review deadline

- Note: reviews are compulsory

Lesson 101: Get your story right

- Many projects contain good ideas, but do not distil what they are.
- Figure out what your story is
  - What's the problem (motivation)?
  - What's the key idea (your solution/approach)?
  - What's the likely benefit/tradeoff?
  - What did you build? Did it work?
  - What did you conclude?
  - Is what you've produced usable/useful?
- Hint: Sketch your story on a piece of paper using simple, clear words. Be 100% explicit.
- Hint: Remember that your project is an investigation - finding out that your idea didn't work is not the same as failing your project



Example story sketch

- Garbage collection is central to modern programming languages, like Haskell
- Real-time Haskell applications need real-time collectors
- Classical real-time collectors rely on read barriers
- Read barriers are traditionally expensive
- We can use GHC's dynamic dispatch to build a really cheap barrier by "hijacking" an object's jump table
- We pay up front for the dynamic dispatch, but then the extra work is minimal
- The payoff is that there is no barrier when the collector is off, and only then when an object has yet to be copied
- We've built an implementation. It works!
- The barrier overhead is found to be small – much smaller than existing schemes
- It looks like the same idea could be used in OO languages, like Java …

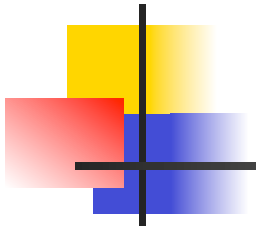
The **Idea**

Example story sketch

- Garbage collection is central to modern programming languages, like Haskell
- Real-time Haskell applications need real-time collectors
- Classical real-time collectors rely on read barriers
- Read barriers are traditionally expensive
- We can use GHC's dynamic dispatch to build a really cheap barrier by "hijacking" an object's jump table
- We pay up front for the dynamic dispatch, but then the extra work is minimal
- The payoff is that there is no barrier when the collector is off, and only then when an object has yet to be copied
- We've built an implementation. It works, but...
- The overhead turned out to be much higher than expected, because...
- All benchmarks ran slower with the new scheme than with the old

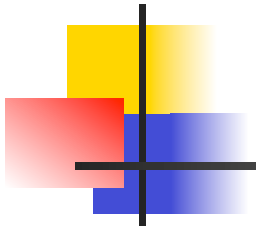
The **Idea**

Oh dear! But we couldn't have predicted this



Report Structure

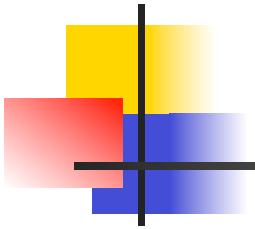
- Title
- Abstract
- Introduction
- Background
- The details
- Evaluation/Discussion
- Conclusions and further work
- Bibliography
- (Appendices)



The abstract

- An extremely condensed summary of the project, comprising a small number of sentences
- Make minimal references to the background
- Focus on the project content
- Hint: write the abstract last

(Not required in the interim report)



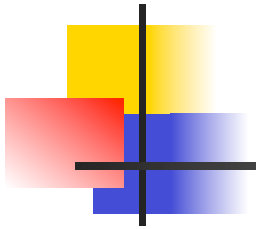
Example (Mark Thomas, 2003)

UNIX desktop environments are a mess. The proliferation of incompatible and inconsistent user interface toolkits is now the primary factor in the failure of enterprises to adopt UNIX as a desktop solution.

This report documents the creation of a comprehensive, elegant framework for a complete windowing system, including a standardised graphical user interface toolkit. 'Y' addresses many of the problems associated with current systems, whilst keeping and improving on best features.

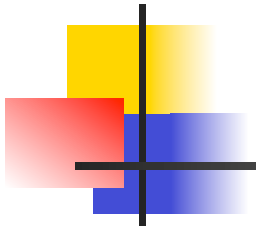
An initial implementation, which supports simple applications like a terminal emulator, a clock and a calculator, is provided.

- This is beautiful, but what's missing?



Example (Will Deacon, 2009)

This project describes and implements a framework to support the execution of specialised methods within the Jikes RVM. The idea is that by applying suitable bytecode transformations at class loading time, method dispatch can be hijacked to invoke a different method implementation depending upon the state of the receiver. These transformations are specified by the programmer using a simple API and are treated independently from the application on which they operate. We evaluate the overheads incurred by our modified virtual machine and motivate the use of class transformations by means of a read barrier and an implementation of transparent object persistence for Java. Our results show that the performance penalty of our framework is extremely low and our technique for specifying class transformations is both efficient and concise.

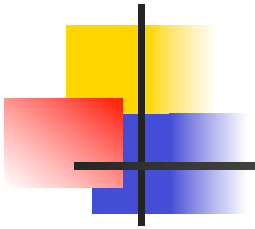


Report Structure

- Title
- Abstract
- Introduction
- Background
- The details
- Evaluation/Discussion
- Conclusions and further work
- Bibliography
- (Appendices)

The introduction

- Tell your story...
  - Draw the reader in: start with the big picture then focus in on the specific issue(s) addressed by the project
  - Avoid philosophical rambling & excessive background
  - Use examples to explain the problem and your idea
- Summarise your contributions
- Provide pointers to key chapters/sections of the report as you go along



Example introduction

Cut to the chase (no philosophy)

1. Introduction

In modern functional program languages, pattern matching is a well-established mechanism for specifying rewrite rules over structured types. The pattern matching order in these languages is traditionally *lexicographical,* which specifies that the function must behave as though the patterns were matched in a *top-down, left-to-right* order.

The lexicographical scheme is easily understood operationally, but it can sometimes conflict with the intuitive reading of a program. As an example, consider the following Haskell function **`boolOR`**, which implements boolean disjunction:

Examples help to clarify the problem

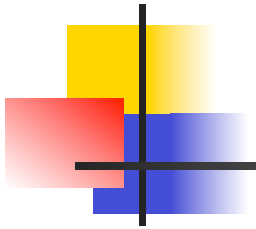
```
boolOR :: Bool -> Bool -> Bool
boolOR False False = False
boolOR x     y     = True
```

Syntactically, this definition is symmetrical in the two arguments, and our intuition is that **`boolOR`** is commutative. However, if one of the arguments is non-terminating (semantic value **`UNDEF`**) then, under the lexicographical pattern matching scheme, **`boolOR True UNDEF = True`**, whereas **`boolOR UNDEF True = UNDEF`**. The symmetry is broken.

The objective of this project is to change the semantics of pattern matching in the Glasgow Haskell Compiler (GHC) so that functions that can terminate always do. With respect to the above example our objective is thus to build an implementation of Haskell for which **`boolOR True UNDEF = boolOR UNDEF True = True`**.

The idea is to implement pattern matching concurrently, by forking separate threads to match different parts of a pattern... Etc.

The idea



The contributions

- The contributions drive the rest of the report: the report substantiates the claims you have made
- Reader thinks “gosh, if they can really deliver this, that’d be exciting; I’d better read on”

Try sketching your likely contributions now

No “rest of the report is...”

- Not:

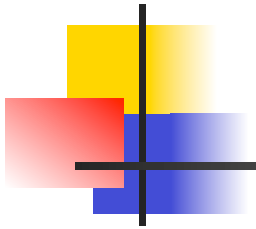
> “The rest of the report is structured as follows. Chapter 2 presents the background. Chapter 3 ... Finally, Chapter 8 concludes”.

- Instead, use forward references from the narrative in the introduction.

Example contributions

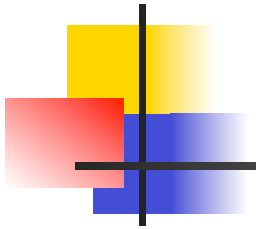
- We suggest a simple technique that exploits GHC's underlying dynamic-dispatch mechanism to support the read barrier invariant (Chapter 2). The key advantage is that there is no barrier overhead associated with object references when either i. the garbage collector is off, or ii. the garbage collector is on and the object has already been scavenged.
- We extend the idea to treat stack activation records in a similar way, so that stacks, too, can be collected incrementally (Chapter 3).
- We detail a prototype implementation for the Glasgow Haskell Compiler (Chapter 4) and highlight some of the key design tradeoffs.
- Using benchmarks from the "nofib" suite, we show that the average overhead on program execution time is less than 4% for the benchmarks tested and that sub-millisecond average pause times are achieved in all cases (Chapter 5).

Make the contributions clear, e.g. by a bulleted list



Final Report Structure (June)

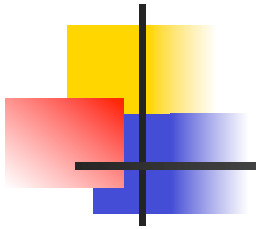
- Title
- Abstract
- Introduction
- Background
- The details
- Evaluation/Discussion
- Conclusions and further work
- Bibliography
- (Appendices)



Background

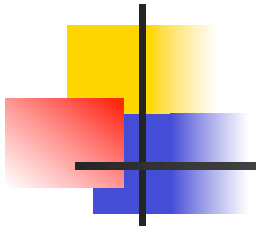
- Make it clear that you're aware of the important key work in the subject
- Provide enough detail to make the report as self-contained as possible, but no more
- Avoid 'routine' background, e.g. the C programming language
- Don't cite sources that are irrelevant or that you haven't read

(You can write most of your background in the interim report)



Report Structure

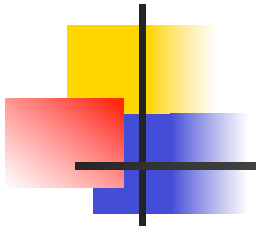
- Title
- Abstract
- Introduction
- Background
- The details
- Evaluation/Discussion
- Conclusions and further work
- Bibliography
- (Appendices)



The details

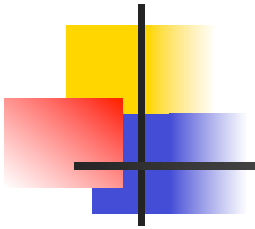
- Say what you did, how and why (“method”); this is highly project-specific
- Justify your decisions regarding design, tools, techniques etc.
- Emphasise the key challenges that you encountered and how you overcame them

(For the interim report outline your planned work and what you’ve done so far)



Intuition first…

- Explain your ideas as if you were speaking to someone using a whiteboard. Only then go into the details.
- Conveying the intuition is primary, not secondary
- Once your reader has the intuition, they can follow the details (but not vice versa)
- Even if they skip the details, they still take away something valuable
- Use examples and diagrams to help



Use examples

This would be extremely hard to explain without the example

3.1 Fast Entry

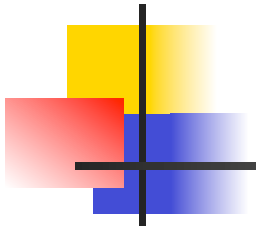
The crucial property on which our scheme depends is that every object is entered before it is used. However, GHC sometimes short-circuits this behaviour. Consider this function:

```
f x = let { g y = x+y } in (g 3, g 4)
```

The function g will be represented by a dynamically-allocated function closure, capturing its free variable x. At the call sites, GHC knows statically what code will be executed, so instead of entering the closure for g, it simply loads a pointer to g into a register, and the argument (3 or 4) into another, and jumps directly to g's code.

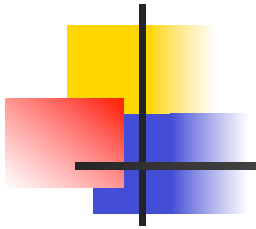
Notice that this only applies for dynamically-allocated function closures. For example, the function f also has a closure, but it is allocated statically, and captures no free variables. Hence it does not need to be scavenged, so calls to f can still be optimised into direct jumps.

Using our approach, the optimisation fails, because we must be sure to scavenge g's closure before using it. The simple solution, which we adopt, is to turn off the optimisation. It turns out (as we show in Section 5) that this has a performance cost of less than 2%.



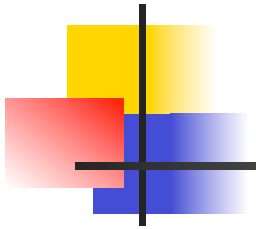
Report Structure

- Title
- Abstract
- Introduction
- Background
- The details
- Evaluation/Discussion
- Conclusions and further work
- Bibliography
- (Appendices)



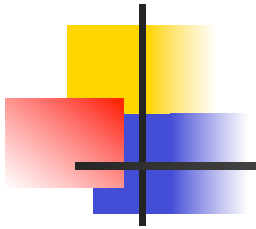
Evaluation

- You must evaluate your successes and failures, bearing in mind what others have already done/ achieved
- Make sure that each of your claimed contributions (in the introduction) is backed up with evidence
- Quantitative evaluation is about performing experiments and making sense of numerical results
- Qualitative evaluation is about making judgements – it is more subjective
- Tip: Try to turn qualitative evaluation into quantitative evaluation



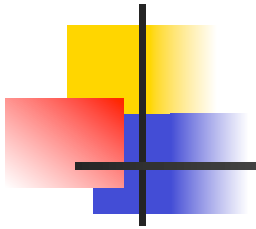
Quantitative evaluation

- Detail your experiment(s)
  - The reader must be able to reproduce your results
- Present results in tables and/or graphs
- Discuss the results and provide insight
- Explain interesting artefacts. Don't leave it to the reader to interpret your results (this is the hard bit!)



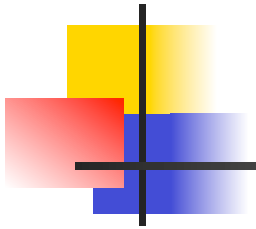
Qualitative evaluation

- Try to relate experience, rather than rely on speculation
  - “Users were able to use the system by just being told the rules of the game…”
  - “The use of shadows gave much better depth perception and made it easier to…”
  - “The need to cast all numerical values to doubles was a universally unpopular feature…”
- Hint: be quantitative wherever possible
  - E.g. size of code, time to solve a given problem, number of mouse clicks…



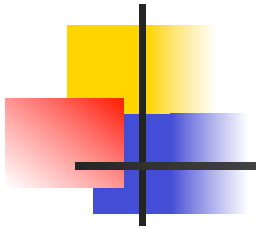
Report Structure

- Title
- Abstract
- Introduction
- Background
- The details
- Evaluation/Discussion
- Conclusions and further work
- Bibliography
- (Appendices)



Report Structure

- Title
- Abstract
- Introduction
- Background
- The details
- Evaluation/Discussion
- Conclusions and further work
- Bibliography
- (Appendices)



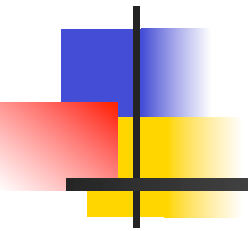
Bibliography

- Cite your sources properly, e.g.

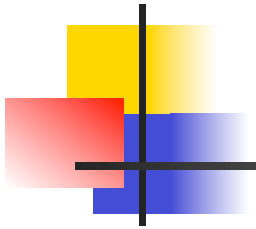
[4] A. Printezis and A. Garthwaite, “Visualising the Train garbage collector”, Proceedings of the Third International Symposium on Memory Management (ISMM’02), ACM SIGPLAN Notices, pp. 100-105, Berlin, June 2002. ACM Press.

[5] The libunwind project, http://www.hpl.hp.com/research/linux/libunwind/

- The reader should be able to locate all your sources of inspiration directly

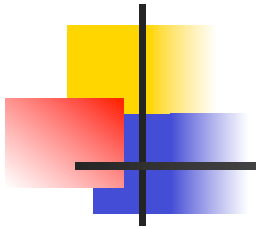


Language and style



Basic stuff

- Submit by the deadline
- A higher page count does not imply more marks.
- Stick to a limit, e.g. binder sizes 75-100, or 100-130 and make it comfortable to read:
  - Do not widen (or narrow!) the margins
  - Do not artificially use big fonts (or very small ones)
  - Use appendices for, e.g. a user guide, experimental data, raw output, detailed proof, etc.
  - Do not include program listings in the report
- Always use a spell checker (**`ispell -t xxx.tex`**)



Visual structure

- Give strong visual structure to your report using
  - sections and sub-sections
  - bullets
  - italics
  - laid-out code
- Find out how to draw beautiful pictures, and use them
- Always refer explicitly to figures and tables in the text

Visual structure

In the STG-machine every program object is represented as a *closure*. The first field of each closure is a pointer to statically-allocated *entry code*, above which sits an *info table* that contains static information relating to the object's type, notably its layout. An example is shown in Figure 1 for an object with four fields, the first two of which are pointers (pointers always precede non-pointers). The layout information is used by the collector when evacuating or scavenging the closure.

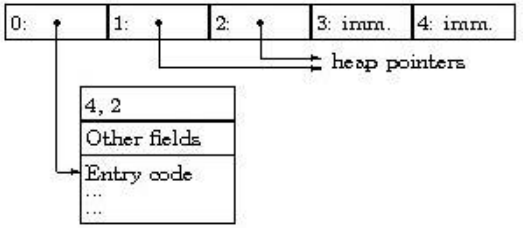


Figure 1: The layout of a closure with four fields, the first two of which are pointers (the other two are immediate values).

Some closures represent *functions*; their entry code is simply the code to execute when the function is called. Others represent *constructors*, such as list cons cells. Their entry code returns immediately with the returned value being the original closure (the closure and the list cell are one and the same). Some closures represent *thunks* (known by some as "suspensions" or "futures"). When the value of a thunk is demanded, the mutator simply *enters* the closure by jumping to its entry-code without performing any test. When the evaluation of the thunk is complete, the thunk is overwritten in-place with a new closure representing its value. If the mutator tries to evaluate the thunk again, execution will now land in the code for the value (i.e. return code), instead of code to compute the value.[1]

The most important feature of the STG-machine for our purposes is that a closure has some control over its own operational behaviour, via its entry code pointer. This representation of heap objects is quite typical in object-oriented systems, except that the header word of an object typically points to a static method table rather than to a single, distinguished method (our "entry code").

[1]Note that some values are bigger than the closures that built them. In these cases the closure is replaced with a pointer to the value (an indirection).

2.3 The New Scheme

The main idea is a cheap implementation of the read-barrier invariant (Section 2.1), which states that the mutator sees only to-space pointers. The STG machine spends most of its time executing code generated from Haskell function definitions. This code executes in an immediate environment, or *activation* consisting of (a) registers, (b) locations in the function's *stack frame*, and (c) the fields of the *currently-active closure* (CAC). This is quite conventional. The CAC is necessary to support first-class functions and thunks: it contains the environment captured when the function or thunk was allocated. (Object-oriented programmers call the CAC the *this* pointer.) The STG machine captures a separate flat (i.e. non-nested) environment in each such closure, a design decision that turns out to make incremental garbage collection much easier.

Since the mutator only accesses the current activation, we can maintain the read-barrier invariant thus:

> *Ensure that all the fields of the current activation have been scavenged; that is, they point into to-space*

In operational terms, there are two major elements to implementing this requirement.

- Before entering the code for a closure, we must first scavenge the closure (Section 2.3.1).
- When a function returns to its caller, we must first scavenge the caller's stack frame (Section 2.4).

2.3.1 Entering a Closure

The first thing that happens to a closure when it is entered during a garbage collection is that it is evacuated and placed on the queue of closures in to-space waiting to be scavenged. At some subsequent

Use the active voice

The passive voice is “respectable” but it can sometimes deaden the tone of your report.

NO	YES
It can be seen that...	We can see that...
34 tests were run	I 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

“I” = you the author

“We” = the authors

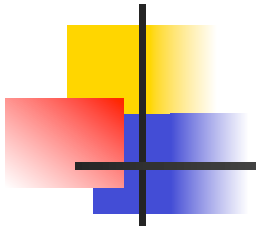
“You” = the reader

Use simple, direct language

NO	YES
This pattern is exemplified pictorially in Figure 5.5	This is shown in Figure 5.5
The object under study was displaced horizontally	The ball moved sideways
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

Don’t use “noise words”

NO	YES
We then simply recompile it	We then recompile it
The actual memory used was never more than 256KB	The memory used was never more than 256KB
It actually turned out that the second approach proved to be the most efficient of the two	The second approach was more efficient
However, of course, this would break the type system	However, this would break the type system



Some other sources

Research Skills (Simon Peyton Jones) (http://research.microsoft.com/~simonpj/papers/giving-a-talk/giving-a-talk.htm) (Thanks to Simon for providing the template for this talk)

Scientific Communication http://undergraduate.csse.uwa.edu.au/units/CITS7200/

Advice on Research and Writing http://www-2.cs.cmu.edu/afs/cs.cmu.edu/user/mleone/web/how-to.html