Exam report for COMP4181/9181 (13s2)

z3416506

Critical assessment of Paper 1.

Problem that the paper tries to address

Jan Bracker and Andy Gill's technical paper, *Sunroof: A Monadic DSL to Generate JavaScript*, attempt to generate JavaScript programs through the domain specific language, *Sunroof*, which is embedded in Haskell. They discuss the usefulness of JavaScript (e.g. graphical canvases, event handling, and first-class functions), but also note that it lacks some desirable features, such as Haskell's static typing.

Bracker and Gill propose Sunroof as an alternative to JavaScript, since Sunroof is able to introduce many of Haskell's features to programmers that JavaScript is unable to natively facilitate (e.g. a threading model, a static type checker, etc.). Since Haskell has an extremely powerful type system, JavaScript programs that are generated through Sunroof are more likely to be correct than if the JavaScript was handwritten.

Sunroof is implemented through a monad similar to the `IO` monad found in Haskell, but uses an extra argument to determine which threading model is to be used. Unlike native, handwirtten JavaScript, Sunroof is able to provide concurrent JavaScript, since it is embedded in Haskell. This is an important step up from handwritten JavaScript, since parallel computations are increasingly becoming important.

Coverage of related work

The authors claim that their work differs from previous research since the previously published papers because:

- Other works do not attempt to create a direct connection between Haskell and JavaScript continuations,
- The Sunroof server provides support for communication with websites requesting code snippets via the *Kansas comet push mechanism*, and,
- Sunroof is the only EDSL to support JavaScript generation inside type-safe Haskell.

Of the thirty references to other works made in this paper, seventeen of these references are explicitly considered to be related in some way to Sunroof. The authors note the similarities of related work, but do not go into great detail about any of them. This is not necessarily bad; there are too many to go into great detail of each, and their level of definition is more than enough to encourage interested readers (with sufficient time) to investigate the related works.

The remaining works do not appear to be directly related to research associated with Sunroof; they are more related to Haskell features used to implement

Sunroof. Consider '*Our example type* `JSString` *has a* `Monoid` *and an* `IsString` *instance that are not provided for other wrappers, e.g.* `JSBool` *or* `JSNumber`*. This approach was first introduced by Svenningsson [29].*' Svenningson and Axelsson had done previous research regarding shallow and deep embedding in reference 29, and Bracker and Gill were able to capitalise on this. They provided reference to a highly detailed technical paper written by Svenningsson and Axelsson regarding the topic[1], which encourages further research should the reader wish to learn more about Sunroof's implementation.

Originality and technical soundness of the underlying ideas

The idea of generating JavaScript through Haskell is not original, as it is mentioned in the Related Work section that *Fay* compiles subsets of Haskell to JavaScript. Bracker and Gill point out that that Sunroof is the only EDSL that generates JavaScript inside Haskell that is type-safe. No evidence has been presented to suggest otherwise at the time of writing this analysis. This doesn't make the process novel. *Accelerate* generates *CUDA* in a similar manner[1].

The only completely novel proposal identified is the Sunroof server previously mentioned. Unfortunately, this section is done little justice, as the content spans for only half a page, including the provided code. While the section is short, the idea presented is really innovative and lightweight in contrast to other web frameworks such as *HAppS*, *Snap*, and *Yesod*, and does not require much explanation; even Haskell novices should be able to infer the code necessary to achieve what is being communicated here.

The paper presents a technical overview of the topic from too high a level. The problem with this is that they do not provide much depth or analysis, and often do not justify why they made such decisions. It would be relatively easy to reproduce the calculator discussed in the paper, and given that the authors have specified how many lines of code are necessary to complete the task, it is possible to evaluate their accuracy, provided that the authors have a sample calculator to test against. Unfortunately, this does not seem the case, as neither mention of the calculator's specification, nor an external reference to such a specification is made in this paper. Additionally, the table provided is not informative, since it does not contrast how many lines of code would be necessary in handwritten JavaScript; although it is possible that the 'Percentage' column alludes to this, insufficient information in the column's heading is provided to prove this.

Evaluation of the presented approach

Overall, the paper is structured well. However, there are a few key things to take note of. Their analysis is not quite as deep as some, and this paper certainly does not make any attempt to motivate using Sunroof over other EDSLs until

[1]The author of this critical analysis didn't have time to properly read this paper, but did read through enough of it to get the gist of what Bracker and Gill were alluding to.

the very end. Additionally, they do not provide any hard evidence to prove that Sunroof is indeed a superior EDSL. This is crucial for implementors as they will need to be convinced that Sunroof is worth their time (and possibly money) learning; this may not be achieved, since there are neither code snippet comparisons, nor anything to quantify that the Sunroof server model is superior to other web frameworks in practice.

Many of the ideas communicated are brilliant and a sense of closure is almost always apparent; however, without the any of the above, it is difficult to believe that this paper has a convincing tone. Perhaps if the paper (or a subsequent paper published by Bracker and Gill) was able to meet any one of the above criteria, it would be able to provide more insight into why Sunroof is a desirable EDSL.

References

1. AccelerateHS, https://github.com/AccelerateHS/accelerate

End of critical assessment of Paper 1.

Critical assessment of Paper 2.

Problem that the paper tries to address

Mio: A High-Performance Multicore IO Manager for GHC, written by Andreas Voellmy, Junchang Wangm, Paul Hudak, and Kazuhiko Yamamoto, addresses a very real concurrency problem. They argue that Haskell threads are key to high-performance, concurrent programs. Although GHC provides support for Haskell threads, its runtime system doesn't natively scale well on multicore processors, and so network applications that try to take advantage of lightweight Haskell threads will perform poorly. The problems that this paper address are:

- Identifying why GHC's RTS performance doesn't natively scale on multicore processors.
- Implementing a way for the RTS to scale on multicore processors.

The authors motivate this problem very well, and explain in great detail the reasons why this problem is one of great concern in the opening paragraphs. While reading the introduction, it is difficult not to consider it to be an extension of the abstract, since the introduction goes beyond its definition and summarises the remainder of the paper.

Coverage of related work

Many of this paper's references are not related to similar works. Works that are related are each discussed very briefly; each receives a few sentences outlining what their work achieves. One such example is, '*The Go programming language [4] provides sophisticated coroutines, called goroutines...*', which does the related work just enough justice to encourage a motivated reader to consider the reference.

Related works discussed toward the end of the related work section are given more time. This is important, as these works require a further expansion to see how closely they are related, and also to show how they differ. They are not just references to other programming languages that implement similar features; they are implementations in arbitrary languages.

Overall, Vollemy et al. discuss related work quite well, and do justify their differences. Any similarities between *Mio* and related work is not discussed.

Originality and technical soundness of the underlying ideas

This paper presents an original idea and is also extremely technical. The paper begins by ensuring that readers are familiar with the GHC threaded runtime system, and how it interacts with the operating system. By not assuming that

the reader has previous knowledge in the field, the authors are able to achieve a greater level of depth when discussing their implementation of *Mio*, since the authors know the minimum level of all readers.

The paper contains lots of Haskell source code that is used to justify their arguments. Where code is not provided in the report, links are provided to code hosted online. The latter is extremely useful, as it allows for further analytical study without taking additional space in the paper. Assuming what is provided is correct, using source code to justify arguments in general is useful, since it proves them in a very concise manner. Correct Haskell programs are even more difficult to dispute, as they are generally contain pure code; it is a lot easier to dispute an imperative program without a rigorous proof than a functional program.

Much of this paper's depth comes from the authors' empirical analysis of systems studied. For example, in *Problem that the paper tries to address*, we learned that the authors discover that one serious bottleneck is due to the somewhat cyclic activity of workers interfering with each other. This analysis in particular is accompanied by three charts:

- one plotting the throughput of the `SimpleServer` as it currently is, and as it evolves over time;
- one describing a single epoch of activity[2]; and,
- one showing several epochs

This is incredibly useful, as it shows the problem close up, but then moves out to show the bigger problem; this illustrates that the problem.

Each problem is then resolved directly after the problem has been identified. Continuing with our *Concurrent Callback Tables* example, the problem is solved as discussed in the first part of this paper; the results are provided in the original throughput chart (this is one evolution), and another chart that shows multiple epochs. There is no need for a single epoch chart, as the situation that required illustration via the single epoch chart is no longer present, and it is much easier to observe thread activity from a distance. This problem-resolution pattern is continued for each problem introduced, showing consistency in their analysis.

Following the analysis on performance, the authors discuss the implementation of *Mio*. This section receives less coverage than the previous one, but still provides a lot of detailed information. Although it provides an excusable reason why *Mio* couldn't be implemented in GHC for Windows, several 'complications' are made note of as 'complications', but not documented. Anyone not familiar with Windows' internals may not receive full closure in this section.

[2] An epoch is a single time step

Vollemy et al. use scientific methods to help prove that program integrity. Section 5 discusses operating system bottlenecks and bugs; to be certain that the implementation wasn't the source of the bottleneck, they implemented a C equivalent of `SimpleServer` as a control[3]. Since both the programs are subject to the same problem (and the two programs are expressed very differently), this eliminates the possibility that the Haskell implementation is at fault. Again, once problems are identified, they are plotted on a throughput chart, where each problem resolution is an evolution. The reader may find it interesting to note that because it was used as a control, `SimpleServerC` also helped to identify a concurrency bug in the Linux kernel.

Most of the previously discussed methods above are again present in the evaluation of *Mio*. In this section, the throughput charts do not represent evolution of a single system, but rather the performance of many systems. *Mio*'s results are impressive, and are discussed well.

Evaluation of the presented approach

End of critical assessment of Paper 2.

[3]Called `SimpleServerC`.

This page is intended to be blank.

The purpose of this page is to prevent accidental scrolling and revealing the identity of the student.

Full name: **Di Bella**, Christopher James
Student number: z3416506

By submitting this report for assessment as the exam component of COMP4181/9181 (13s2), I declare that this submission is my own work, and I have not received any help whatsoever.