Literature Review

The potential of declarative programming languages to support user interface programming: the case of ELM

Simon Buist

Bachelor of Science in Computer Science
The University of Bath
October 2013

Contents

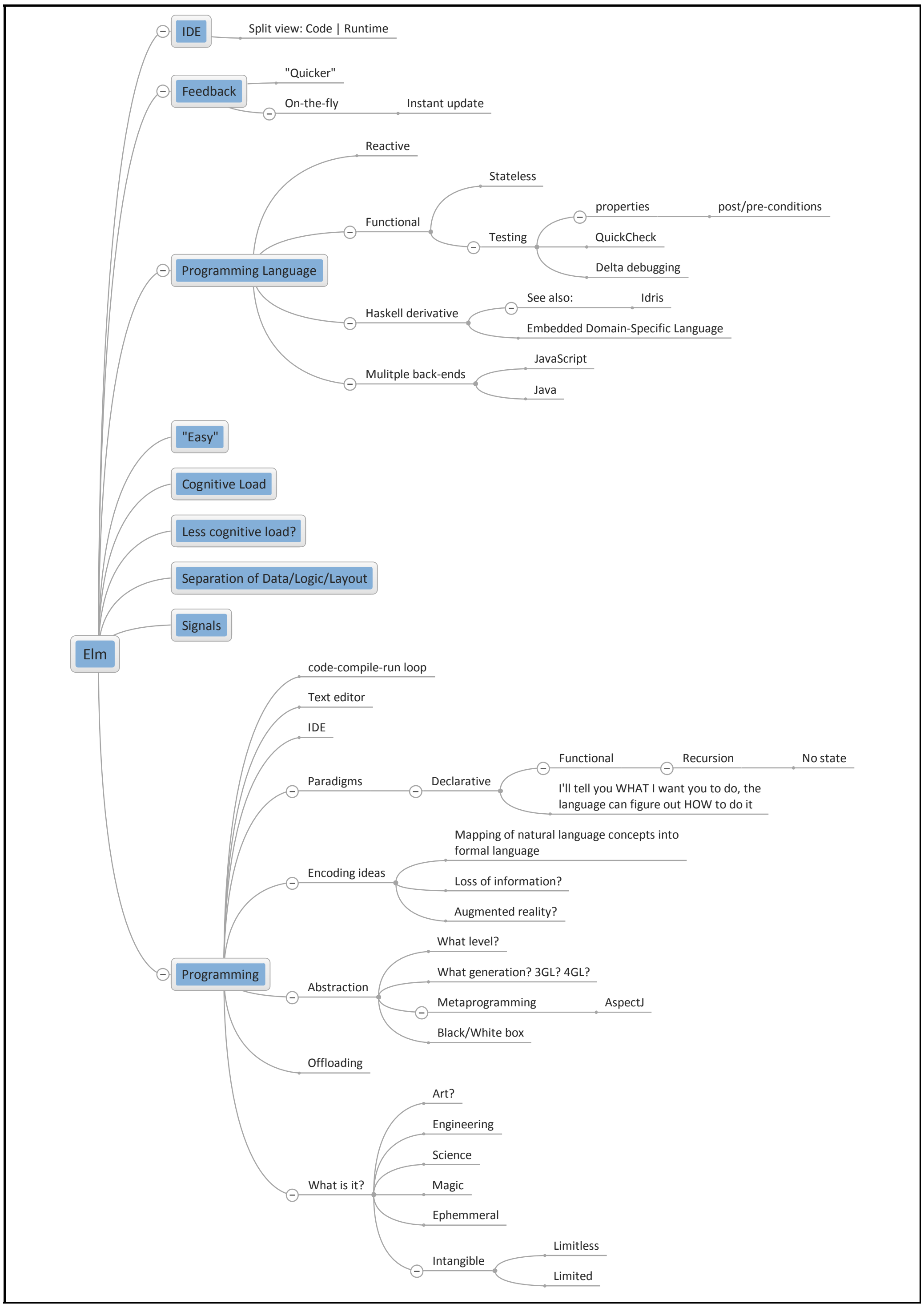
Chapter 1

A survey of relevant papers

1.1 Introduction to the problem area

Spending half an hour making a mind-map, starting with the word “Elm”, I get the following terms:

1.1.1 Elm mind-map



In list form, with added terms:

- Signals
  - Impure
  - Time
  - Input/Output
  - History – Past & Future
  - Data model
- “Natural” Separation of Data model/Logic/Layout
- Less cognitive load?
- “Easy”
  - Professed to be “easy” by the creator!
  - What does it mean to be “easy”?
  - Operationalisation
  - Self-reporting
- Programming Language
  - Reactive
    - “Instant feedback”
    - “I/O-sensitive”
    - Thrashing?
  - Functional
    - Pure
    - Testing
      - properties
      - QuickCheck
      - Delta Debugging
  - Haskell derivative
    - Embedded Domain-specific Language
    - See also: Idris
  - Multiple back-ends
    - Javascript
    - Java
    - C

- Programming
  - code-compile-run loop
    - “Programming blind”
    - “Slow feedback”
  - Text editor
  - IDE
  - Paradigms
    - Declarative
    - I’ll tell you WHAT I want you to do, you figure out HOW to do it
  - Encoding ideas
    - Mapping of natural language concepts into formal language
    - Loss of information?
    - Augmented reality?
  - Abstraction
    - What level?
    - What generation? 3GL? 4GL?
    - Metaprogramming e.g. AspectJ
    - Black/White box
  - Cognitive offloading
  - What is it?
    - Art?
    - Engineering?
    - Science?
    - Language?
    - Mathematics?
    - Ephemeral
    - Intangible
    - Limitless
    - Limited
- IDE
  - Split View: Code — Runtime
- Feedback
  - “Instant-update”
  - On-the-fly

The problem area of user-interface programming, and more generally, the activity of programming in a context such as a software engineering environment, encompasses certain realms of interest. Through my survey of literature, my research has touched upon the above-mentioned terms, and I have discovered some thought-provoking problems that exist in the field of programming. The concept of 'Programming' embodies other concepts – art-forms, engineering processes, science, language, and mathematics, among others. To me, programming is a creative endeavour unlike any other – in which the programmer weilds materials of no substance – the code – by manipulating symbols on a screen, which represent states in the machine being used. There are so many programming languages, and all languages (all that are Turing-complete) reduce to the same language – that of a Turing Machine. So, *why do we have so many programming languages?.*

Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy. (Perlis, 1982)

Different languages lend themselves to different ways of thinking about problems. They may place emphasis on one feature, for example list manipulation and hide others such as types. The language or programming environment may make explicit the effect of changes as they are encoded, as opposed to queuing up a block of changes and the programmer having to initiate an update manually.

I would like to draw your attention in particular to the terms **Abstraction**, **Cognitive offloading**, **Feedback**, **Loss of information?/Augmented reality?**, **Thrashing**, and **"Programming blind"**. These, at current, are my topics of interest, and my literature review has up to this point been inextricably and heavily influenced by this.

Under the umbrella quote of "Don't program blind" – tools that help the programmer see what he's doing:

From: [Don't Learn to Code, Learn to Program – But come back in 10 years](http://johnkurkowski.com/p
learn-to-code-learn-to-program-but-come-back-in-10-years/):

Quite a sad title really. And sad content. BUT it does link to some wise words from Jeff Atwood, Peter Norvig ("Learn to program in 10 years!"), and has a handful of links to different projects and a nice quote:

If builders built buildings the way programmers wrote programs, then the first woodpecker that came along would destroy civilization. Gerald M. Weinberg, The Psychology of Computer Programming (1971)

[Out in the open: These Hackers want to give youSuperpowers](http://www.wired.com/wiredenterprise/2
table/)

We consider programming a modern-day superpower. You can create something out of nothing, cure cancer, build billion-dollar companies, he says. Were looking at how we can give that super power to everyone else.

The problem with coding, he says, is that you cant see the results of your work until after youre done. In that sense, programming is unlike almost every other craft. When a chef adds an ingredient, he can smell it, he can taste it, Granger says. When an artist makes a stroke on a canvas, he can see it. But programming isnt that way.

We consider programming a modern-day superpower. You can create something out of nothing, cure cancer, build billion-dollar companies.

Chris Granger

Programmers may spend hours or days working on code before they can make sure it actually works. We have to play computer in our heads, he says. We write each line, imagine what it will do. You have to act like a computer. The problem with that is that were pretty crappy computers. But Light Table seeks to bridge that gap.

Light Table is an open source programming tool that lets programmers see the results of their code as their write it. Its not an entirely new idea. In the mid-1960s, an educational tool called Logo gave programming students immediate feedback. More recently, languages like the kid friendly Scratch and artist friendly Processing have offered a kind of visual feedback, giving coders more insight into their programs as theyre written.

But applying those ideas to professional software complex applications with thousands or even millions of lines of code is another matter. Light Table tackles such software by not only by displaying the results of the code youre working on right now, but by showing how it relates to other parts of your software and how data flow from one chunk of code to another. It also weaves documentation throughout the code, while offering new ways to organize and visualize the code in any application.

Flow Based Programming

* [NoFlo](http://noflojs.org/), [Flowhub.io](http://flowhub.io/) * BOOK: [J. Paul Morrison](http://www.jpaulmorrison.com/fbp/) – Flow Based Programming * [Example of NoFlo implementation of Jekyll](https://github.com/the-grid/noflo-jekyll)

1.2 What does it mean to be 'easy to use?'

In the process of surveying relevant (and sometimes irrelevant) literature to this dissertation, recurring conceptual patterns were observed – one particular instance of this is that several authors seem to lay victim to the trap of claiming their creation is "easy to use", "better", "simpler than x" without providing any supportive evidence of this.

Perhaps these are incidents of 'experimenter bias' – where the evaluator is naturally predisposed to a positive appraisal of their own findings. One way to avoid this is to have one set of people perform the data capture and another set perform the data analysis. Nevertheless, these patterns emerge, and present numerous opportunities for experimentation and subsequent evidence supporting or contradicting these claims. Experiments may see if the same conclusions are reached as the above-mentioned authors, accounting for the 'evaluator effect' (Hertzum and Jacobsen, 2001).

Whether this particular route is taken for experimentation hinges on pilot studies that will be conducted concurrently to the Literature Survey, each inextricably shaping the other's direction of investigation and inquiry.

The catalyst to this whole dissertation was a talk about the concept of a highly reactive development environment – where changes in the code result in instantaneous updates to the runtime, 'on-the-fly'. This was presented in Bret Victor's "Inventing on Principle" (Victor, 2012). In his presentation Bret makes several assertions about the 'traditional' style of coding, one statement of which is that "most of the developer's time is spent looking at the code, blindly without an immediate connection to the thing they're making". He argues that "so much of creation is discovery, and you can't discover anything if you can't see what you're doing" – alluding to his earlier statement that the compile-run-debug cycle is much like this.

Evan Czaplicki, in his thesis of which Elm is the product (Czaplicki, 2012), makes similar claims – "[Elm] makes it *quick and easy* to create and combine text, images, and video into rich multimedia displays." While the evaluation of Elm's usability is not the focus of the thesis, rather, it is to establish a context for Functional Reactive Programming and describe the implementation details, he makes other usability claims without evidence – "[non-declarative frameworks for graphical user interfaces] mire programmers in the many small, nonessential details of handling user input and modifying the display.", "FRP makes GUI programming much more manageable", and in a section entitled *The Benefits of Functional GUIs*, "In Elm, divisions between data code, display code, and user interaction code arise fairly naturally, helping programmers write robust GUI code". If these claims are true, there is all the more evidence that Elm should be a language of choice for GUI programmers, but experiments must be done to determine this.

And perhaps this rapid development cycle is not always suitable – in their 2012 paper, Lopez et al. show that novices tend to "thrash" about, trying out many ideas that may or may not be a solution, and executing "poorly directed, ineffective problem solving . . . failing to realise they are doing it in good time, and fail to break out of it", whereas experts think much more about the problem at hand before proceeding with a solution (Lopez, Petre and Nuseibeh, 2012).

1.3 Running User Studies

Perhaps a further direction of investigation may be running an experiment to spot whether or not Elm's auto-updating IDE lends to a lack of critical thinking – some operationalization may be *pauses reported as 'thinking' made during development* – where a pause is disambiguated as 'thinking' by the experimenter asking the participant why they did not perform any interaction with the computer for more than 10 seconds, and the participant reports that they were planning/designing/other similar activity. Along this line of thinking, a paper studying the relationship between speech pauses and cognitive load (Khawaja, Ruiz and Chen, 2008) found through studying 48 mixed gender participants that there is statistically significant indicators of cognitive load through analysing pauses in speech. Perhaps this concept of pauses can be applied to the activity of programming. However, the planned method of disambiguating pauses via self-reporting (previously mentioned) would not be suitable according to these authors – "such measures can be either physically or psychologically intrusive and disrupt the normal flow of the interaction", although a paper cited by (Khawaja et al., 2008) itself claims that "although self-ratings may appear questionable, it has been demonstrated that people are quite capable of giving a numerical indication of their perceived mental burden (Gopher and Braune, 1984)". Indeed a pilot study by Fraser and Kölling (Mckay and Kölling, 2012) structures the self-reporting by getting the users to evaluate an IDE as they use it using a set of subject-specific heuristics that they have designed. They showed that this customised set of heuristics helped guide the user more effectively than Nielsen's heuristics in evaluating usability, so one could develop a custom set of heuristics for evaluating the usability of Elm.

From the Elm thesis (Czaplicki, 2012), the language syntax and rapid feedback seem simple enough that it is conceivable (or at the very least, possible and of experimental interest) to allow the user to customise the UI layout to their liking. Letting the user shape the UI in concert with a UI programmer is covered the study of the interface development environment "Mobi-D" in millitary and medical applications (Puerta, 1997), with success in those fields. It may be worth speculating how Elm would fit into the development cycle that Puerta's paper outlines, as this may lend inspiration to potential user interface enhancements to the Elm IDE for A/B testing. It must be noted that there does not seem to be a re-emergence of Mobi-D since the paper was written, however.

My goal is to answer these questions. By way of conducting user studies, leveraging Elm with extensions to do A/B testing to illustrate it's effectiveness (or ineffectiveness) at enhancing User Interface Design.

Central to this idea of iteration is my desired method of performing user studies: I will first do what I have called a "Pilot" – a short and shallow trial User Study that focuses not on the research I'm concerned with, but instead the particular experimental design I would like to use in my actual User Study. By employing a Pilot I can hopefully get an idea of the nature of the experimental design – perhaps discovering any variables I had not previously considered that will require me to increase my sample size or simplify the experiment in order to mitigate their effect on the dependent variable I wish to test for. These are all problems discovered in (Yates, 2012) – including basic teething problems in getting the experiment to flow smoothly. In an even less detailed aspect, the pilot may allow me to look at what is out there. It may help to not look for anything in particular initially, and see what happens.

At this stage, with the help of discussion with my Project Supervisor, I have some ideas about how to gather data in User Studies and these pilots could prove to be a useful testbed for such tools. I have a hypothesis that the novice developer "thrashing" (Lopez et al., 2012) can be observed by shorter pauses between editing and experimentation, and I could measure this by way of measuring the mouse position relative to the IDE, clicks, and key-presses, using tools built-in to Elm and a bit of extension to stream this over the

Internet to my storage facilities (Czaplicki, 2013).

Bibliography

Czaplicki, E. (2012), ‘Elm: Concurrent frp for functional guis’.

Czaplicki, E. (2013), ‘What is functional reactive programming?’.
URL: http://elm-lang.org/learn/What-is-FRP.elm

Gopher, D. and Braune, R. (1984), ‘On the psychophysics of workload: Why bother with subjective measures?’, *Human Factors: The Journal of the Human Factors and Ergonomics Society* **26**(5), 519–532.

Hertzum, M. and Jacobsen, N. E. (2001), ‘The evaluator effect: A chilling fact about usability evaluation methods’, *International Journal of Human-Computer Interaction* **13**(4), 421–443.

Khawaja, M. A., Ruiz, N. and Chen, F. (2008), Think before you talk: An empirical study of relationship between speech pauses and cognitive load, *in* ‘Proceedings of the 20th Australasian Conference on Computer-Human Interaction: Designing for Habitus and Habitat’, OZCHI ’08, ACM, New York, NY, USA, pp. 335–338.
URL: http://doi.acm.org/10.1145/1517744.1517814

Lopez, T., Petre, M. and Nuseibeh, B. (2012), Thrashing, tolerating and compromising in software development, *in* Y. Jing, ed., ‘Psychology of Programming Interest Group Annual Conference (PPIG-2012)’, London Metropolitan University, London Metropolitan University, UK, pp. 70–81.

Mckay, F. and Kölling, M. (2012), ‘Evaluation of Subject-Specific Heuristics for Initial Learning Environments : A Pilot Study’, (November), 128–138.

Perlis, A. J. (1982), ‘Epigrams on programming’, *SIGPLAN Notices* **17**(9), 7–13.

Puerta, A. R. (1997), ‘A model-based interface development environment’, *IEEE Softw.* **14**(4), 40–47.
URL: http://dx.doi.org/10.1109/52.595902

Victor, B. (2012), Inventing on principle, Presented at the Canadian University Software Engineering Conference (CUSEC).
URL: http://vimeo.com/36579366

Yates, R. (2012), Conducting field studies in software engineering: An experience report, *in* Y. Jing, ed., 'Psychology of Programming Interest Group Annual Conference (PPIG-2012)', London Metropolitan University, London Metropolitan University, UK, pp. 82–85.