Title
McMaster University

Jason Balaci

Submission Date

Abstract

todo

Acknowledgements

todo

Contents

Chapter 1

Introduction

1.1 Context: Knowledge Capture & Encoding

TODO: While my MSc focus isn't necessarily exactly "When Capturing Knowledge Improves Productivity", I should have a reasonable amount of discussion here about it, or else readers might question the need for my work entirely. I guess I would also need a bit of justification for that programming ideology as well.

As a software developer working to build a new piece of software, one might find that they are writing "a lot of the same code" as their or other pre-existing software projects. One might consider building a library, shared between all projects for some common functionality/tooling, this is a large improvement for their program — being able to reuse their code is great for debugging and removing the possibility of bugs when stable and tested code is used.

Commonly, one looks to use mature libraries and frameworks to underpin their projects, occasionally without guarantee that connecting these libraries is safe. As general purpose programming languages are often also used, misunderstandings of tacit project knowledge may also cause errors. Unfortunately, this programming methodology takes significant time until a working product is formed with minimal bugs.

If one originally sets out to build a program that does something they understand very well and each component of every step of the grand scheme of the program was understood, the development of their related software

should be a clear matter of principled engineering. However, with this existing programming methodology, it is not yet simple enough yet to get reliable programs.

Optimally, they would use their natural language to perfectly describe their problem to their computer and have it magically give them a program that does exactly what they wanted. Unfortunately, it is difficult to have computers systematically understand and act on natural language as well as us humans can. However, we can mimic a computer "understanding" an equivalent speech through having the human write the same knowledge using a domain-specific language. Additionally, through sequencing and connecting domain-specific languages, we can effectively model what they are trying to build, allowing the computer computer to be able to better understand what they are trying to build. With enough effort, the computer should be able to generate some artifact that ultimately represents what they were trying to build. Note, however, that this idea likely only thrives in domains of knowledge that is "well-understood".

TODO: Discuss the domain-expert's job of creating the domain-specific languages and tools, and the importance of their works containing everything important

With a focus on building Scientific Computing Software, Drasil is an exploration of this idea. Rather than building one's software project in any single or combination of general purpose programming language, the usage of a sequence of domain-specific languages together in Drasil-based projects can be used to describe common undergraduate level physics models and problems and describe the a target program that simulates said models.

1.2 Problem Statement

TODO: Clean up below

As a domain expert transcribing knowledge encodings of some well-understood domain, one will largely be discussing the ways in which pieces of knowledge are *constructed* and *relate to each other*. In order for this abstract knowledge encodings to be usable in *some way*, it is vital to have "names" (*types*) for the knowledge encodings. In working to capture the working knowledge of a domain, it's of utmost importance to ensure that all "instances" of your "names" (types) are *always* usable in some meaningful way. In other words, all knowledge encodings should create an stringent, explicit

set of rules for which all "instances" should conform to, and, arguably, also creates a justification for the need to create that particular knowledge/data type. As such, optimally, a domain expert would write their knowledge encodings and renderers in a general purpose programming language with a sound type system (e.g., Haskell, Agda, Java, etc) – preferring ones with a type system based on formal type theories for their feature richness.

In Drasil, we are focused in understanding families of scientific software, and creating systematic rules to generate families of software solutions (for any instance of a scientific problem that requires a scientific software solution). Specifically, Drasil is focused on mathematics and physics-based models. In both areas, we are concerned with what *kinds of theories* are wellunderstood, and ensuring that all created mathematical expressions are "valid". TODO: Still a bit awkward

1.3 Thesis Outline

todo

Chapter 2

Background

2.1 Project Focus & Goals

todo

2.2 In practice/Architecture

todo

2.2.1 Chunks

todo

2.2.2 ChunkDB?

todo

2.3 Methodology for locating and encoding knowledge – e.g., "bottom-up" approach

todo

2.4 State of Drasil

todo

2.4.1 Short-term problems – leading into topics

todo

Chapter 3

Topic #1: Typing the Expression Language

3.1 Background: Problem

todo

3.2 Requirements & properties of a good solution

todo

3.3 Solution

todo

3.3.1 Expression encodings discussion (GADTs, TTF)

todo

3.3.2 Dividing expression languages (CodeExpr, Expr, ModelExpr, Literals)

todo

3.4 Continued problem with Expressions – leading into ModelKinds

todo

Chapter 4

Topic #2: Framing Theories in Contexts

4.1 Problem

todo

4.2 Requirements & properties of a good solution

todo

4.3 Solution – ModelKinds

todo

4.3.1 EquationalModels

todo

4.3.2 EquationalRealms

todo

4.3.3 EquationalConstraints

todo

4.3.4 DEModel

todo

4.3.5 Continued

todo

Chapter 5

Future Work

todo

Chapter 6

Conclusion

todo

Appendix A

Appendix

todo