Title Text

Subtitle Text, if any

Dan Szymczak
McMaster University
szymczdm@mcmaster.ca

Jacques Carette Spencer Smith
McMaster University
carette@mcmaster.ca / smiths@mcmaster.ca

Abstract

This will be filled in once we're done drafting. For now I'm putting in a solid block of text that will let me ramble on for a while when it comes down to the final abstract. Programmers are like gods of tiny universes. They create worlds according to their rules. If we think of software artifacts (that is, all documentation, build instructions, source code, etc.) as their own tiny worlds, a universe takes shape. Now think about what would happen if the laws of that universe were inconsistent between worlds. What if the laws of thermodynamics fundamentally changed as soon as you left Earth? Say goodbye to space travel. Keeping artifacts consistent with each other is hugely important in the software world, and Drasil aims to ensure it...

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

Keywords Literate software, knowledge capture, traceability, software engineering,scientific computing, artifact generation, software quality

1. Introduction

We have a vision to make better software. We want to increase software's maintainability, traceability, reproducibility, verifiability, and reusability. Our vision is to invest more in the short-term to provide outstanding long-term benefits and is where we see the fundamental trade-off in our technique: it is the complete opposite of agile-development while simultaneously throwing away the waterfall model.

We intend to make working on program families trivial for the cost of losing the capability to create "local hacks" to "just get things working".

Before we get into more detail on the technique, let's start with an obvious disclaimer: it would be a pipe dream to claim we could use this technique over all software domains. In fact, the domain(s) we will be looking at are very restrictive, but highly useful. We have restricted the scope of our work to only those areas of software with a *well understood* knowledge base.

We will be sticking to using Scientific Computing (SC) software for illustrative purposes throughout this paper, but the fundamental ideas (and, in fact, our framework) should be applicable to any well understood software domain.

We believe Literate Programming (LP) had the right idea, but in our opinion is far too restricted. Rather than just coupling the source code with its documentation, we want to build all of our software artifacts (requirements and design documents, source files, test cases, build instructions, etc.) from a singular common knowledge base.

We want to break away from the LP idea of "one source", instead introducing the idea of "common knowledge" libraries for problems that come up regularly. For example: the theory behind heat transfer including the conservation of energy equations would be in a "common knowledge" library. We could also see the idea of the Système International (SI) Units as another piece of "common knowledge".

We also very much want to keep the idea of "chunks" and assemble everything from chunks. How we do that will be discussed in more detail in Section 2.

Another disclaimer before we go further: none of the ideas presented in this paper are new. We are building on a very longstanding history of work, and not being particularly creative with the ideas therein. Knuth himself even said that "I have simply combined a bunch of ideas that have been in the air for a long time" when he coined LP (Knuth 1984). We are, however, taking old ideas and breaking them down into something practical that we can work with. We want to *do* something, not spend years stuck in the design phase.

Others have done similar work (which we will get into in Section **??**), but they did not achieve the results we are envisioning. They either set their aims on other targets, spent too long creating a grand design and ended up without any

real, practical results, or they simply were not brave enough to break things down into the smallest necessary chunks.

We believe that we have assembled the right ideas to achieve our vision. The overall success or failure of our approach hinges on the idea of a stable, well understood knowledge base. Without that we might as well give up and go home. If we do not understand the fundamentals behind the software we intend to create, then we will be unable to properly encapsulate that fundamental knowledge for use in our framework (effectively getting us nowhere).

As it stands, we are on a (not so) humble, practical route to achieving our goals and improving the overall quality of software.

2. Drasil

Our framework is called Drasil (shortened from *Yggdrasil* from Norse Mythology, which is also known as the *world tree*). It is currently being developed through a practical, example-driven approach in an effort to bring our vision to life. There are three main ideas driving Drasil's development:

1. Organize the knowledge base – We want a knowledge base that we can structure conceptually (i.e. keep knowledge for a certain class of problems together). This is where chunks come into play: each chunk encapsulates a single piece of knowledge like a concept or a quantity. We want to break our knowledge base down into the smallest possible pieces.
2. Use recipes to create artifacts – We can think of each artifact in our software project as a different view of our knowledge base. We want to use recipes to specify exactly what information from the knowledge base is necessary for each artifact, and how that information should be displayed. For many artifacts we would like to have a standard recipe which can be quickly customized for the current problem and that is how we will avoid duplicating knowledge.
3. Remove technology constraints – We want to be able to create our software without worrying about the underlying technical constraints of our display or specification technology, programming language(s), etc. Anyone using Drasil should be able to work with the knowledge base and their recipes, then simply set their output technology and have the generator take care of all the technical details.

We argue that by implementing Drasil around these ideas, we can see drastic improvements in software quality. In fact, using a generative approach we can avoid certain problems altogether. One obvious and recurring problem that comes to mind when upgrading software is "feature creep" or "software bloat" (Amsel et al. 2011). With Drasil, a software upgrade will actually be a completely new piece of software that includes the previously desired functionality as well as the new upgrades. We will discuss further improvements in more detail after our example in Section 2.2.

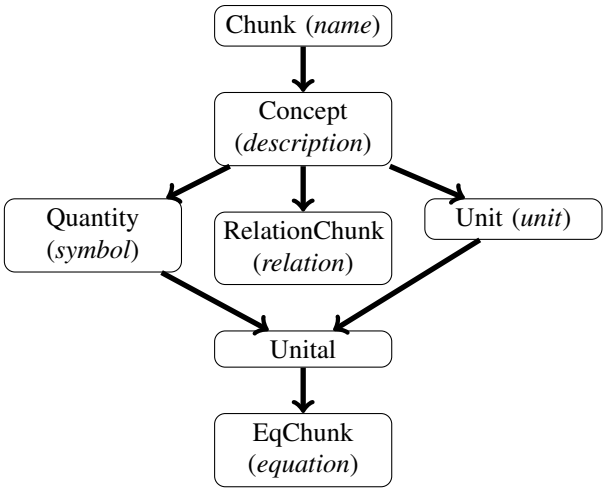

Figure 1. Our current chunk design

2.1 Drasil's current implementation

Fundamentally, Drasil must be able to capture knowledge and produce different views of that knowledge. With our current implementation we have each individual piece of knowledge as a named *chunk*. We are then able to manipulate our chunks through the use of a *recipe*. Our *generator* interprets the recipes to produce the final desired view. This view represents one of the many software artifacts mentioned in Section 1.

As mentioned, we capture knowledge into named chunks, of which there are several varieties. Actually, we have a hierarchy of chunk types, where each new chunk encapsulates more than the previous one(s) (see Figure 1 for an idea). The most basic *chunk* represents a named piece of information. Above that we have named *concepts* which introduce some descriptive information.

A *quantity* is a concept that has a symbolic representation, we can refer to the quantity by either its name or symbol. In a similar vein are *units*, and *relation chunks* (which add the idea of a relation between some other pieces of knowledge).

In an SC context, most of the knowledge we work with is represented as a quantity with some units, in other words a *unital* chunk. Expanding on that, we can actually calculate values for many of these unital concepts. As such, we have *eqchunks* which allow us to capture the equation along with everything else included in a unital chunk.

Now with the means to encapsulate knowledge, we can turn our attention to our recipes. The recipes are implemented using a combination of embedded Domain Specific Languages (DSLs) in Haskell. We have currently implemented the following DSLs:

1. Expression language – A simple expression language that allows us to capture knowledge relating to equations and mathematical operations. It includes, but is not limited to, operations such as addition, multiplication, negation, derivation, and exponentiation.
2. Expression layout language – A micro-scoped language for describing how expressions should appear. Expressions may need to use subscripts, superscripts, concatenated symbols, etc. to be properly displayed.
3. Document layout language – A macro-scoped language for describing how large-scale layout objects (tables, sections, figures, etc.) should appear.
4. C Representation Language – A DSL for representing parts of the C programming language inside the Drasil framework. This allows the generator to produce working C code.
5. LaTeXRepresentation Language – A DSL for representing LaTeXcode inside of Drasil.
6. HTML Representation Language – A DSL for representing HTML within Drasil.

Of the DSLs mentioned, we only actually have to write our recipes using three of them. Each of the representation languages are used strictly by the generator as an intermediary in the production of our desired views. We write our recipes using the document layout language, expression layout language, and expression language. We will discuss the particulars of each of these in more depth during our example (Section 2.2).

The last piece of the puzzle is the generator. We use it to interpret the recipes, create intermediary representations of our desired views, and then pretty-print them.

2.2 Drasil in Action

For this section we will take a look at a simplified version of a Software Requirements Specification (SRS) for a fuel pin in a nuclear reactor (for more information on that particular SRS see (Smith and Koothoor 2016)).

A. Appendix Title

This is the text of the appendix, if you need one.

Acknowledgments

Acknowledgments, if needed.

References

N. Amsel, Z. Ibrahim, A. Malik, and B. Tomlinson. Toward sustainable software engineering (nier track). In *Proceedings of the 33rd International Conference on Software Engineering*, pages 976–979. ACM, 2011.

D. E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984. doi: 10.1093/comjnl/27.2.97. URL `http://comjnl.oxfordjournals.org/content/27/2/97.abstract`.

S. Smith and N. Koothoor. A document driven method for certifying scientific computing software used in nuclear safety analysis. *Nuclear Engineering and Technology*, Accepted, 2016. 42 pp.