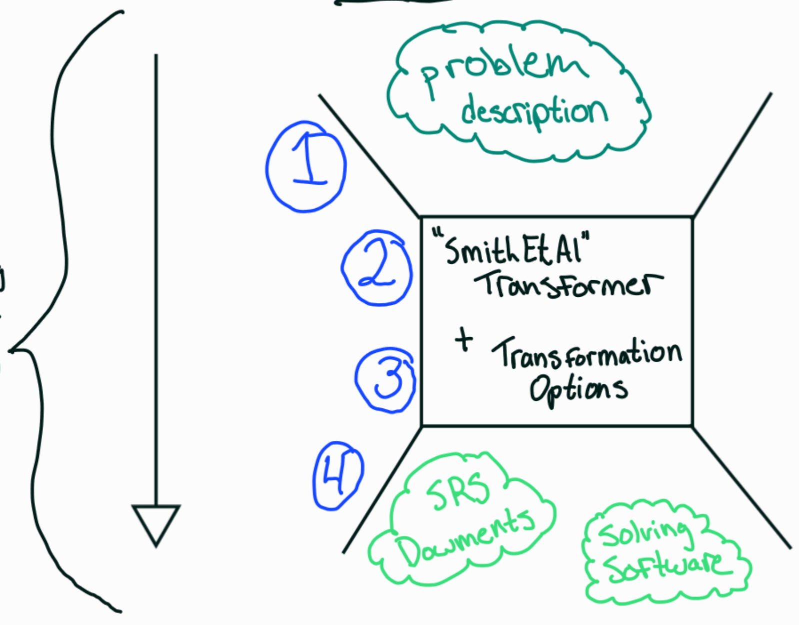
PhD Topic Proposal #1 : DraDrasilsil

Drasil is a software artifact[1] generation suite that takes a "Knowledge capture"[2] approach to building software artifacts, whereby an input set of information is systematically manipulated until a "final, desirable" information (i.e., artifacts) are formed. Drasil is currently used to generate a series of physics-problem solving software artifacts and coherent descriptions of the physics "problems" using a structured "Software Requirements Specification (SRS)" document and conventional physics and mathematics syntax and semantics.

Figure 1

Each of Drasil's case studies use this general flow to building software artifacts. A coherent problem description (1) is created using a precise structure and then fed in to a compiler/transformer (2) that systematically derives output artifacts (4), with a bit of "flavour choices" consciously made (3).

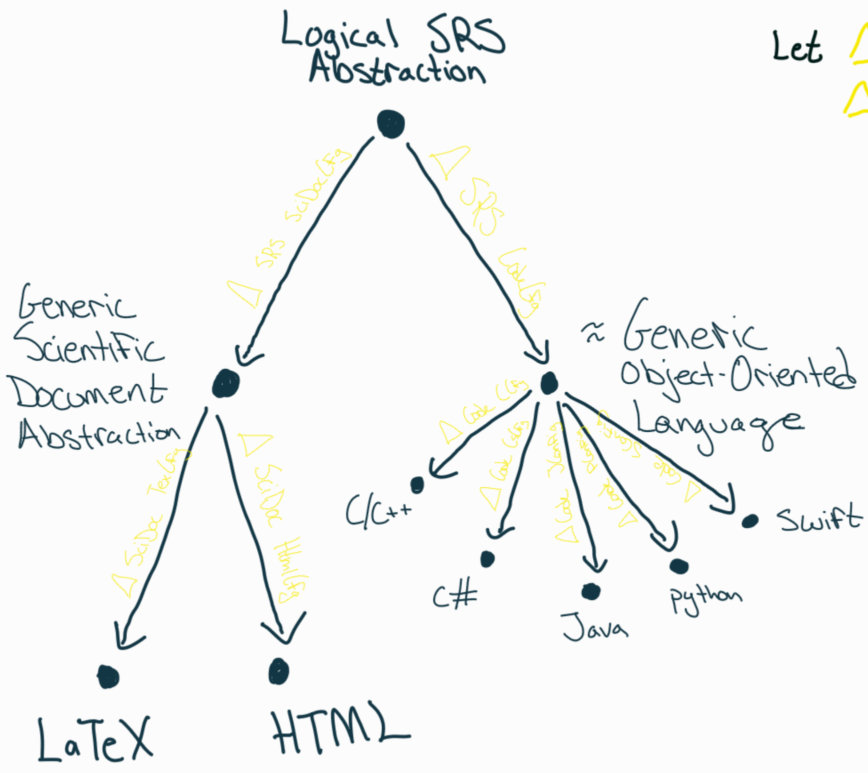


1 "software artifact" is intentionally ambiguous, it is any single, or collection of, artifacts that are somehow usable on your computer (e.g., anything!).

2 Recently, I've been enjoying calling this a "well-derived" approach.

Figure 1 depicts Drasil's largest compiler, a multi-source compiler, the SmithEtAl transformer. However, it should be made clear that Drasil is more than "just" that. For example, the well-derived approach implies that the transformer can be dissected into it's logical steps, and it indeed can be! Each step or subset of steps is also a compiler that can be used within Drasil. For example, we can dissect Figure 1's depicted compiler further into Figure 2.

Figure 2: A slightly closer look at the SmithEtAl transformer.



Let $\Delta : \text{Artifact}_{i} \to \text{Artifact}_{cfg} \to \text{Artifact}_{o}$

$\Delta = \lambda\ \text{in}, \text{opts} \to$ "transformer that takes an input artifact and a transformation option set, and forms an output artifact"

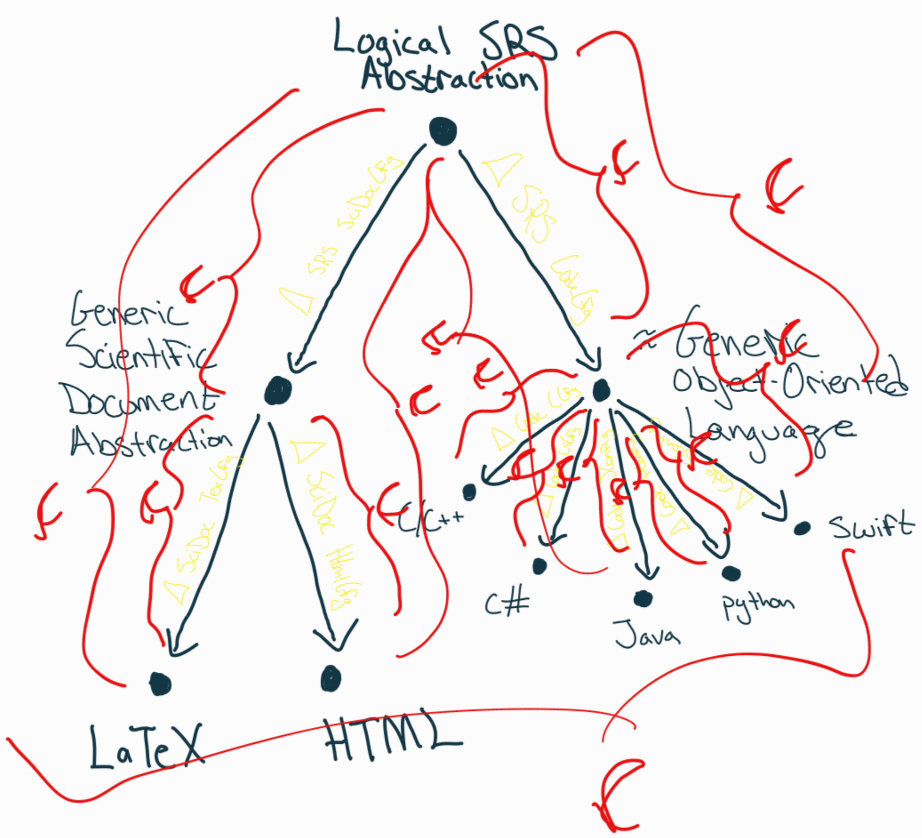
(NOTE: green text represents a type parameter)

With Figure 2, we may observe that the larger transformer consists of a composition of many, smaller, transformers[3], Figure 3.

[3] Note that each of the "smaller" transformers has it's own config. options. The transformers that use these smaller ones must somehow "roll-up" or fulfill these other internal config. options.

Figure 3

Let $\mathcal{C}$ = "Compiler"

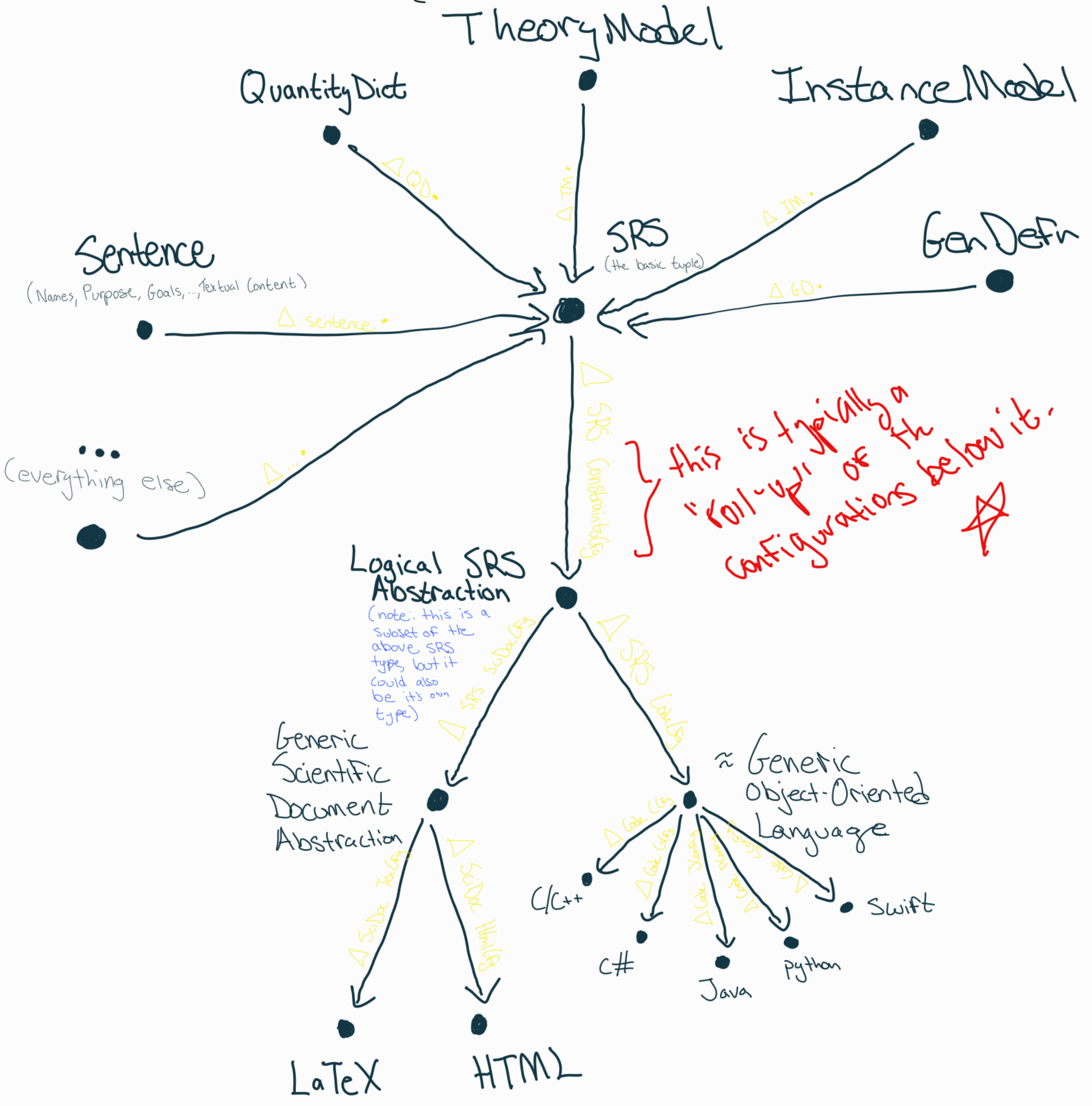

Logical SRS Abstraction
Generic Scientific Document Abstraction
≈ Generic Object-Oriented Language
LaTeX
HTML
C/C++
C#
Java
python
Swift
$\mathcal{C}$

Any line you trace in this is another compiler you can use.

Figure 3 should make it a bit more clear that the SmithEtAl transformer compiler relies on many others, Each of the other compilers may also be used on their own or as part of another, larger transformer. Figure 3 also shows how the "knowledge capture", or "well-derived", or "generative domain model", approaches improve knowledge reusability (and explicitly at that!). Notably, each node represents a concrete, large, chunk that represents a "whole" artifact. However, each node shown here actually has it's own network of sub-transformers &

Chunks. We do not need to worry about those for our purposes. However, we do need to talk about the flow of information in Figure 2/3: Figure 4, where each of the "top" nodes are user-provided inputs to the SmithEtAl transformer.

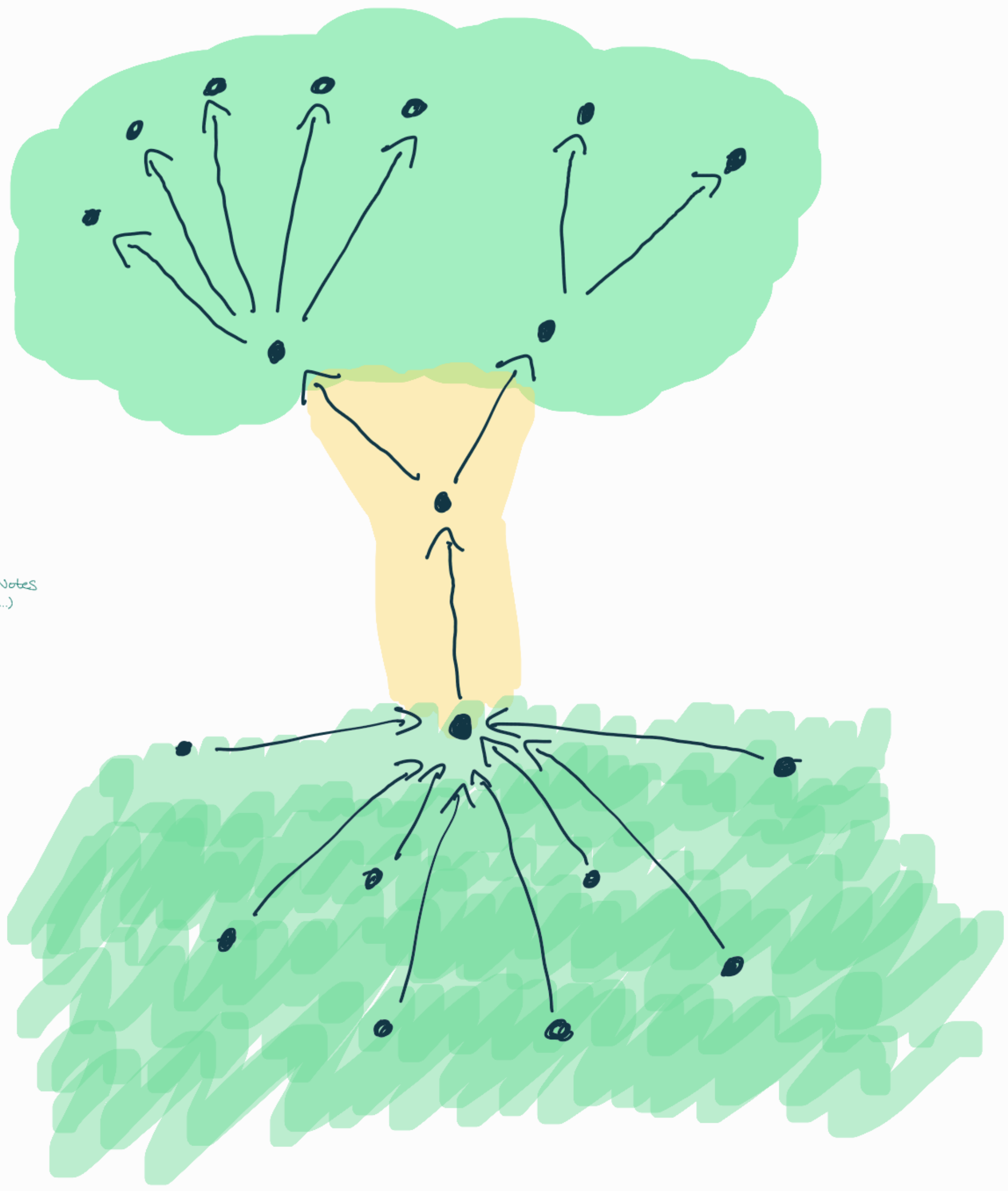
Figure 4



Drasil exercises the harmony formed by this network of domains. Figure 4 shows how multiple source languages (the nodes) are connected using transformers and how a harmony of information synchronization, origination, and communication is formed between multiple different languages. Funny enough, Figure 4 displays one of Drasil's many possible "trees"... you just need to flip the figure.

Figure 5: SmithEtAl Artifact Flow Tree

(I had to re-draw this because Samsung Notes apparently lacks the ability to rotate drawings...)



In some sense, Drasil is a "forest" of trees.

Thus far, all of the figures shown have been manually built. How can we automate the formation of these figures & more tools (ISSUE #1)?
In order to automate this construction, Drasil needs a means of discussing these bodies of knowledge. Currently, Drasil is deeply embedded in Haskell. To generate one of these figures we would need to treat the "transformers" and all chunks as external bodies of information we can work with.

where are the components shown in Figure 1?

(1), (2), & (3) are all deeply embedde in Haskell code. These 3 bodies depend on Drasil's core libraries, (4) is an interesting oddity. It is also deeply embedded in Haskell code, but is externalized (on the host system) as it is the "final" software artifact that developers are interested in building. Thus, it should not really be thought of as any different from the other 3; it has a working Haskell encoding but is externalized. All 4 bodies depend on types that the Drasil core libraries provides. (1) is primarily user-provided instances (think "cookie cutter" data) of said types. (2) is presumed to be part of Drasil libraries, but users may also extend it, in which case, (1), (2), (3), & (4) may potentially rely on external Haskell code.

The general flow of Figure 1 is also approximately how each case study program works. Each "Drasil program" has the same deeply embedded structure:

i. Collect user-inputted chunks, validating constraints along the way.
ii. Operate on the pool of seeded chunks using the directed transformer and defined configuration.
iii. Dump the output artifacts (if applicable) on the host system.

Although each case study has this general flow, it is not made explicit through Drasil formalizations and introspecting Drasil programs automatically is an arduous process involving GHC internals, reflection, and other Haskell metaprogramming tools. As a result, forming compositions of Drasil programs is difficult without Haskell connectives as well (ISSUE #2). In other words, Drasil's understanding of Drasil is shallow.

So, how are Drasil programs tested & executed? As everything is deeply embedded in Haskell, everything outside of Haskell's type system is "checked" at Drasil's runtime. This means that debugging Drasil programs is difficult because it is only done after Haskell compilation (ISSUE #3).

Interestingly, GHC is a sophisticated compiler capable of partial evaluation. Other than system-originating timestamps, every piece of data in Drasil -- all chunk types, chunks, transformers, & configurations. Thus, using Haskell is dubious, the whole Drasil program should be partially evaluated to nearly the extent of "full evaluation" minus side-effects. This brings in to question whether Drasil should be a compiled language or not (ISSUE #4).

All of Drasil's case studies are based on manually constructed software artifacts which are dissected & analyzed for re-creation as the output of a generator. For the case studies, this works great to excavate general knowledge, improve reusability and improve the general quality of software artifacts. However, in doing so, we are building Drasil manually. Has there been enough constructed such that we can learn how to re-generate Drasil? What would it look like? (ISSUE #5).

THINGS TO WRITE ABOUT

5. Can we build tooling to support the creation of IDE-like tooling for working with languages/encodings built in Daffodil.

← or automate!

$\downarrow$

6. Working with Haskell is arduous when we update a chunk type & have a systematic approach to "Fixing" the instances that use it. Is there any way we can build highly generic but extensible chunk refactoring tooling?

$\downarrow$

7. Should Haskell even be the entry method of Chunks in to Drasil?

$\downarrow$

8. How can de-coupling the Drasil "program" From the Drasil program allow us to improve accessibility to software usage and development?

↓

10. How does Drasil bring down the barrier to entry of software without diminishing the quality of software?

↓

12. What are the two kinds of programs Drasil works with?
    ↳ case studies (generated)
    ↳ Drasil (manually built)

↓

13. What is the difference between Drasil & a case study? Nothing! Ole!
    How can we rebuild Drasil in Drasil?

↓

14. #3194 is a great example of something we should be closer to generating.

Think about the Smith EtAl transformer...

the "template" project is something that should be completely generated for us.

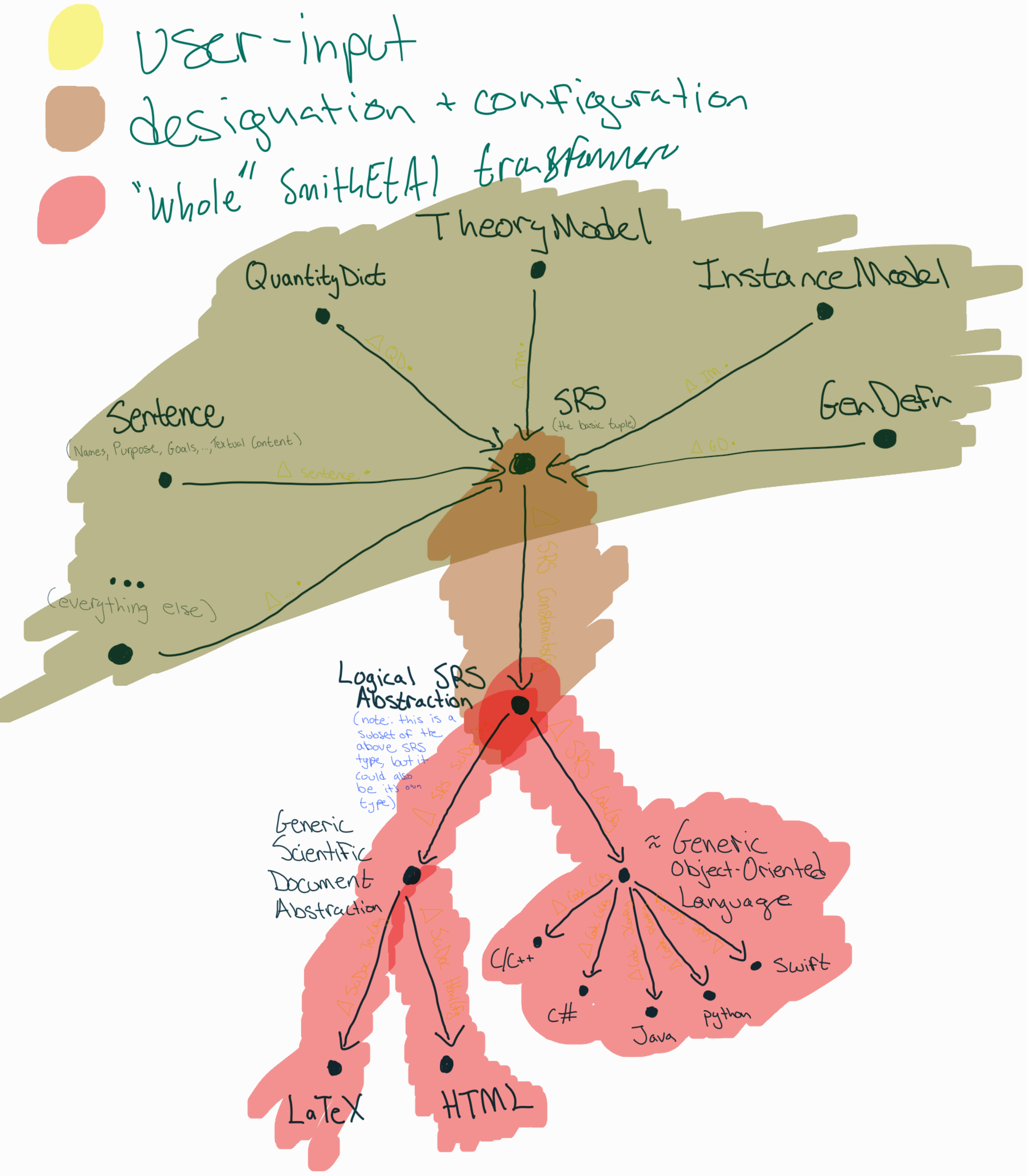
APPENDIX

A.1

The coherent SRS abstraction follows the SRS template defined by Smith.

$$\text{Let } SRS = (\text{Name}, \text{Authors}, \text{Purpose}, \text{Goals}, \ldots, \text{Quantities}, \ldots \text{Theories})$$

Note: each of the above are approximately types (e.g., plural names are lists of the things).



User-input
designation + configuration
"Whole" SmithEtAl transformer
TheoryModel
QuantityDict
InstanceModel
Sentence
(Names, Purpose, Goals, ..., Textual Content)
SRS
(the basic tuple)
GenDefn
...
(everything else)
Δ QD.
Δ TM.
Δ IM.
Δ GD.
Δ sentence.
Δ ....
Δ SRS ConstraintsCfg
Logical SRS Abstraction
(note: this is a subset of the above SRS type, but it could also be it's own type)
Δ SRS SciDocCfg
Δ SRS CodeCfg
Generic Scientific Document Abstraction
≈ Generic Object-Oriented Language
Δ SciDoc TexCfg
Δ SciDoc HtmlCfg
LaTeX
HTML
Δ Code CCfg
Δ Code CsCfg
Δ Code JConfig
Δ Code PConfig
Δ Code SConfig
C/C++
C#
Java
python
Swift

Things to Discuss

1. Partial evaluation & interactivity

2. Making Drasil extensible

3. Removing boilerplate code

4. Analysis & transformation tools

Analysis:

i. graphical display

ii. traceability

iii. general tooling

iv. generating the figures!

Transformation tools:

i. Composition

ii. encouraging re-use

iii. generating IDE tools

iv. generating GUI builders

... analysis is just a specific kind of transformation.