Todo list

ADDING TYPES AND THEORY KINDS TO DRASIL

ADDING TYPES AND THEORY KINDS TO DRASIL

By JASON BALACI, B.Sc.

A Thesis Submitted to the School of Graduate Studies in Partial Fulfillment of the Requirements for the Degree Master of Science in Computer Science

McMaster University

Master of Science (2022)

Hamilton, Ontario (Department of Computing and Software)

TITLE: Adding Types and Theory Kinds to Drasil

AUTHOR: Jason Balaci, B.Sc.

SUPERVISOR: Dr. Jacques Carette

PAGES: xv, 86

Lay Abstract

A lay abstract, with a maximum of 150 words, explaining the key goals and contributions. See https://gs.mcmaster.ca/app/uploads/2019/10/Prep_Guide_Masters_and_Doctoral_Theses_August-2021.pdf for an explanation of the requirements.

lay abstract

Abstract

An abstract, with a maximum of 300 words. See https://gs.mcmaster.ca/app/uploads/2019/10/Prep_Guide_Masters_and_Doctoral_Theses_August-2021.pdf for an explanation of the requirements.

abstract

Acknowledgements

acknowledgements

Contents

List of Figures

List of Tables

List of Source Codes

List of Abbreviations and Symbols

ADT	algebraic datatype
API	application programming interface
AST	algebraic syntax tree
CMS	content management system
DSL	domain-specific language
FFI	foreign function interface
GADT	generalized algebraic datatype
GHC	glasgow haskell compiler
GOOL	generic object-oriented language
HGHC	Heat Transfer Coefficients between Fuel and Cladding in Fuel Rods
HTML	hypertext markup language
IDE	integrated development environment
KMS	knowledge management system
LSS	literate scientific software
ODE	ordinary differential equation
OO	object-oriented
PDF	portable document format

SCS	scientific computing software
SRS	software requirements specification
SSP	Slope Stability analysis Program
SWHS	Solar Water Heating System
TTF	typed tagless final
UID	unique identifier
DblPendulum	Double Pendulum
GamePhysics	Game Physics
GlassBR	Glass Breaking
NoPCM	Solar Water Heating System Without PCM
PDController	Proportional Derivative controller
Projectile	Projectile
SglPendulum	Single Pendulum

Declaration of Academic Achievement

Declaration of academic achievement. See McMaster's thesis guidelines for a more descriptive instruction.

Chapter 1

Introduction

Writing Directives

- *Based on the content of a video by Dr. Cecile Badenhorst (https://www.youtube.com/watch?v=c2oGY1c51jc) and a post about writing introductions by UNSW: https://www.student.unsw.edu.au/introductions*
- Move 1: Establishing a research territory by:
  - showing research area is important, interesting, and incomplete
  - reviewing previous research
- Move 2: Establishing a niche by noting gaps in previous research.
- Move 3: Occupying the niche by:
  - outlining purpose
  - listing research questions
  - announcing principal findings
  - stating the value of the previous research

- General Structure:
  - Introduction:
    - Jazzy information to get reader hooked
    - States purpose of chapter
    - Roadmap of what will be discussed in chapter
  - Background: context of research problem, sets up the need for research and relevance
  - PPSQ: should be within first 3 pages of thesis, after intro + background information.
  - Research design and context: description of where the research takes place (Drasil), introducing methodology briefly
  - Assumptions, limitations, scope of research, and expected outcomes: what do we need from this work
  - Overview of chapters
- Last Paragraph: summarize key points of chapter, link to next chapter
- What is the context of this research? What is it about?
- What problem does this research tackle?
- Why is the research problem important/significant?
- What previous research exists?
- What is the purpose of this research? What are the goals?
- What did the author contribute? *Section 1.4*
- What does this thesis contain? *Section 1.5*

When developing software, developers pull information from a common pool

of domain-specific knowledge, including the requirements and related fields (such as physics and financial mathematics). The pool is common, but not necessarily formally shared, leading to issues in synchronization, communication, and understanding, all of which harm the final software produced in some way. As knowledge changes, software artifacts are slow-moving to adjust, requiring continued development. For well-understood domains, such as Scientific Computing Software (SCS), where there exists commonly agreed upon "knowledge" and software is somehow usable, Drasil aims to capture domain-specific knowledge and use a generative approach to build software as *view* of the captured knowledge. Drasil is a suite for *generative software development*.

Currently focusing on SCS, Drasil uses a stable scientific knowledge-base to generate families of SCS conforming to a Software Requirements Specification (SRS) [1]. By formalizing the language associated with an software requirements specification (SRS) document (such as symbols, units, mathematical expressions, theories, etc.) and how the language terms translate into other languages, Drasil is capable of generating SCS conforming to a software requirements specification (SRS), improving consistency, traceability, understandability, maintainability, and reusability of the scientific computing software (SCS) (but, more importantly, of the domain-specific knowledge as a whole, as well).

Specifically, in this thesis, we will explore capturing mathematical theories frequently pulled from in building SCS, typing mathematical expressions, restricting mathematical expression usage to appropriate contexts, and scaling Drasils knowledge database.

From Dr. Carette: The usual means of building software involves multiple artifacts (such as specifications and code) that contain duplicate information that is also supposed to be linked (*traceability*). Drasil aims to use a generative approach to de-duplicate this information and make traceability more immediate. Drasil currently uses a stable scientific knowledge-base to generate families of Scientific Computing Software (SCS) conforming to a Software Requirements Specification (SRS)[1].

1.1 Background

Intended to "generate all the things"[1], Drasil[2] is a Haskell-based [2] software suite studying how knowledge capture may improve modern software development. "Knowledge" is considered "captured" in Drasil by codifying it and its relations to other things using Domain-Specific Languages (DSLs) encoded as Haskells data types. For example, "quantities" (`QuantityDict`s), as we've currently needed from a physics-based point-of-view, are encoded as:

Source Code 1.1: Definition of "Quantities" (QuantityDicts)[3]

```
data QuantityDict = QD { _id' :: IdeaDict
                       , _typ' :: Space
                       , _symb' :: Stage -> Symbol
                       , _unit' :: Maybe UnitDefn
                       }
```

And an instance of a "quantity" might appear as:

[1] "Generate All the Things" is Drasils tagline.

[2] https://jacquescarette.github.io/Drasil/

[3] https://github.com/JacquesCarette/Drasil/blob/9c26b43d3e30c3f618e534a3f176a5152729af74/code/drasil-lang/Language/Drasil/Chunk/Quantity.hs#L16-L20

Source Code 1.2: Example Encoded Quantity: Tolerable Load[1]

```
tolLoad = vc "tolLoad" (nounPhraseSP "tolerable load")
    (sub (eqSymb dimlessLoad) lTol) Real
```

The ways that we can translate "quantities" into other "things" are encoded as Haskell-level functions and instances of typeclasses, such as:

Code snippet: example of how QuantityDicts are converted into at least 1 or 2 other things (e.g., SRS rows, symbols, etc.).

Drasil is developed through a "bottom-up" methodology against several SCS case studies (Table 3.1), capturing and de-duplicating knowledge as needed to re-generate the original artifacts and more, in a wide variety of similarly applicable languages. For example, Drasil makes use of the similarities of Object-Oriented (OO) languages by forming expressions in Generic Object-Oriented Language (GOOL) before compiling to the specific instances of OO languages (such as Java, C/C, C#, Python, etc.). Similarly, textual markup can be exported in similar languages, such as HTML and LaTeX.

For example, the Glass Breaking (GlassBR) case study (examining predicting whether a glass slab can withstand an explosive blast) had software artifacts[2] manually written. A coherent net of knowledge/discussion (a "story") is then formed by dissecting the artifacts, understanding why each piece existed, what it relates to and how, and how it can be translated into other things. The original artifacts are then re-generated[3] in a wider variety of other languages by codifying how the net of knowledge can be translated into other languages.

However, not all the case studies are capable of generating software artifacts yet (Table 3.3). Each for their own reason, but we will focus on a critical common

[1] https://github.com/JacquesCarette/Drasil/blob/9c26b43d3e30c3f618e534a3f176a5152729af74/code/drasil-example/Drasil/GlassBR/Unitals.hs#L261-L262

[2] https://github.com/smiths/caseStudies/tree/master/CaseStudies/glass

[3] https://github.com/JacquesCarette/Drasil/tree/master/code/stable/glassbr

denominator between them all: capturing mathematical knowledge for reliable SCS artifact generation (and more).

1.2 Problem Statement

Domain-specific abstractions are what enable *domain-specific interpretation and transformation* (e.g., optimization, analysis, error checking, tool support, etc.) [3], which is what sets DSLs apart from general-purpose programming languages. Drasil relies on a *network of domains* (a network of domains of knowledge that somehow relate other domains of knowledge in their own) to enable the complex informational transformation needed to convert an SRS document into code.

As Drasil focuses on the generative software development of SCS, capturing the domain of scientific knowledge is at the forefront of its priority list. The depth and breadth of the network of domains is directly related to how many opportunities for domain-specific transformations we can perform on a specific pool of knowledge.

I feel like I should be able to cite Czarnecki2005 again here, but this isn't really how he worded it, is it fair to cite him here, or should I provide my own justification?

Drasil currently relies on a single universal untyped mathematical language to describe general mathematical knowledge (including, but not limited to, equations, relations, theories, derivations, constraints, and definitions). As general-purpose programming languages are to domain-specific languages, a single universal mathematical language is to structured theory knowledge. In other words, Drasils reliance on a single mathematical language is limiting because of the complexity associated with interpreting its terms, which lack information about the *domain*.

An exercise in futility!

As a result, transforming encoded theories into other forms (such as code) is a complex task (similar to the complexity associated with transpiling a general-purpose program into another). Drasil is unable to make intended/appropriate usage of encoded mathematical theories because of a lack of structural mathematical information. Additionally, as Drasil lacks type information about its mathematical expression language, invalid expressions are possible to be written causing further

issues in reliable formation and transformation into usable high-quality SCS.

Finally, as Drasils network of domains continues to grow in different aspects (such as creating type parameters), Drasil faces difficulties in scaling its database of knowledge against these large changes.

1.3 Research Questions

RQ1 Drasil has a language of simple mathematical expressions that are used in multiple contexts. But not all expressions are valid in all contexts. How do we fix that?

RQ2 Drasil's current encoding of "theories" are essentially black boxes. We would like to be able to use some structural information present in the short list of the "kinds" of theories that show up in scientific computing. How do we codify that?

RQ3 How can we ensure that our language(s) of simple mathematical expressions admits only valid expressions?

RQ4 Our current "typed" approach to collecting different kinds of data is hard to extend. How can we make it easier to extend?

1.4 Contributions of the Author

In listed code snippets, I will refer to at least two major points of time in relation to Drasils development (time measured by their git blob hash): "current"[1] and "original"[2]. The "original" code refers to a code snippet as it was written before I was onboarded to Drasil. The "current" code includes my work, at least, but might also include the work of others who were also contributing to the project while I

[1]Blob hash: dc3674274edb00b1ae0d63e04ba03729e1dbc6f9
[2]Blob hash: 9c26b43d3e30c3f618e534a3f176a5152729af74

was actively contributing. The work of others might include, but not limited to, code formatting, code commenting, and extensions.

Drasil has existed since 2014, and has already seen success in its case studies, which are used to guide the development of Drasil. Drasils focus on SCS relies on knowledge of mathematical theories and language, for which Drasil has a working understanding of before this work. However, some case studies were unable to participate in code generation due to a lack of flexible theory information (RQ2), or just being inapplicable. This work contributes to structuring theory information and allowing for future developers to encode more kinds of theories and their relationships with other things (discussed in Chapter 4). The solution builds on a prototype by Dr. Jacques Carette that facilitates structured theories to define relationships between "code" and "theories."

Cite Dr. Carettes ModelKinds prototype.

Theories rely on mathematical expressions as well. We commonly differ the usable set of language in different contexts (you are free to write a lot more on your pencil and paper derivations than on your typical calculator). To obtain information about the expressibility in different contexts, we divide the expression language using a TTF [4] encoding, with a GADTs backend for structural edits (Chapter 4). However, "expressibility" also relies on the expressions adhering to a precise syntactic set of rules. As such, we build a system of typing rules for the expression language (Chapter 5).

Finally, to enable capturing data with type parameters and generally scale Drasils knowledge database (RQ4), this work merges the typed database collections into a single untyped, yet type-preserving, database (discussed in Chapter 6).

1.5 Thesis Outline

In Chapter 2, we discuss the focal ideology underpinning this work and Drasil. Chapter 3 describes Drasil, the host project carrying the fruits of this work. Chap-

ter 4 discusses how theories are encoded in Drasil (RQ2), the issues associated with using a single universal mathematical language to describe theories (RQ1), and how we can resolve these problems. Chapter 5 describes issues associated with the formation of mathematical expressions (RQ3). Chapter 6 focuses on how Drasil stores information, and how it can be scaled.

Chapter 2

Ideology

I used the passive voice a lot in this chapter. I need to fix that.

The focus of this work is fundamentally based on the idea of "generation." However, unlike GitHub and OpenAI's Copilot [5], the ideology does not delve into artificial intelligence, and does not focus on "autocompleting code" by natural language. Instead of working with the set of natural language, the ideology focuses on formalizing the meaning of specific subsets into principled stories. The principled stories are then intertwined and mixed until a coherent "whole story" is formed, where we deterministically understand what can and cannot be done with the knowledge described (including, but not limited to, generating representational software artifacts and snippets).

2.1 On Developing Software

As a software developer working to build a new piece of software, one might sense 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. The library will surely save them time in the future once they've stabilized it to a reasonable degree, after which they will not need to worry about repeating the same errors they made when they were originally writing it. They will have made significant gains in their development environment and workflow. However, the library might not be portable across machines, the library might not make sense to those more or less familiar with certain ideas touched upon by the library, others might question the validity of the knowledge it pulls from, and the library might not be readily accessible to those not using the same programming language the library was written in. They can write an FFI, but this is a fairly complex task that many are unfamiliar with, and which requires meticulous analysis to ensure compatibility and creates time expensive update procedures when foreign libraries are updated.

Commonly, one looks to use mature libraries and frameworks to underpin their projects, occasionally without guarantee that connecting these libraries is safe. A familiar example of this failure is the sinking of the Vasa [6], partially caused by different teams working together but using different "feet" units (the Swedish foot is 12" while the Amsterdam foot is 11") resulting in unintended weight distribution. As general purpose programming languages are often also used, misunderstandings of tacit project knowledge may also cause errors. Unfortunately, despite building on the shoulders of giants, this programming methodology takes significant time and stress until a working product is formed with minimal bugs.

To resolve this, we believe we need to revisit the original sensation felt when code re-creation/duplication was recognized – why does this sensation occur?

The reason is obvious (ignoring the more obvious fact that they are aware of similarities between their code and that previously written)! They feel this sensation because they already understood it and had written it previously. Implicitly, when writing the software, the developer already had a mental connection between some algorithm and the code they had written (at times, specific to the

programming language they had chosen to use) – it might have already been *well-understood* to them.

Unfortunately, a non-trivial amount of time is spent manually recreating the same piece of code from the same implicit knowledge-base, and the developer is likely left with less than their preferred amount of time to work on the components that are interesting/important that they might never have built before or that they might not have connected together in the past.

2.2 Thoughts of Generation

The ideology foundational to this work is predicated on this simple idea that if we understand how a *thing* works, and we can create a working model describing it, then we should be able to encode it, describing everything about this model. Furthermore, if we understand how this *thing* can be transformed into another *thing*, then we should be able to describe the process of transformation as well. Finally, if we are able to do this en masse to a pool of *various things*, we should be able to generate whole pools of *other various things* from empty, or minimal seedling pools.

Applying this idea to the world of software, where we can create and model *things* (henceforth referred to as "knowledge"), we can take pools of knowledge and generate other pools of knowledge (including whole software artifacts). With a weak breadth and low depth of captured knowledge, the originating pool of knowledge may approximately appear as a description of the software artifacts generated. However, with a strong breadth and high depth of captured knowledge, the original pool of knowledge may appear far removed from the software artifacts generated (it may contain little to no discussion of the desired software artifacts at all, or a precise description). In this case, the stress of software development is alleviated entirely, and a "perfect" software artifact (or series of "perfect" software

Discuss triform theories

artifacts) is (are) constructed.

2.3 The Goal

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/algorithm of the program was understood, the development of their related software should be a "clear matter of principled engineering". However, with the existing programming methodology, it is not yet simple enough to reliably produce error-free programs with no trade-offs, and which precisely satisfy a set of requirements and follow a formal design.

Cite GitHub README? I really don't remember where I read this.

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, specific subsets of natural language may be usable.

2.4 Reconciliation — "Generate Everything"

While it is difficult to have a computer act on a huge set of natural language, we do have well-understood thoughts on specific *terms* and how they can be interpreted. More specifically, there are sub-languages of natural language that we can use to describe specific kinds of knowledge, and problems and solutions. Here, we recognize that selecting or creating domain-specific languages for specific buckets of knowledge and creating domain-specific languages that connect buckets of other domain-specific languages, we can effectively create rigid sub-languages of our natural language used to describe programs. These rigid sub-languages being encoded as domain-specific languages will have a well-defined and formalized AST, which

allows us to write interpreters that can take them and transform them into other fragments of knowledge (us being most interested in ultimately generating software artifacts). Thus, with enough effort, and through sequencing and connecting terms of domain-specific languages, we can effectively model what the discussed software developers are trying to build, allowing the computer to be able to better understand what they are trying to build.

Note, however, that this idea likely only thrives in domains of knowledge that is "well-understood" [7].

In theory, this should appear as a Knowledge Management System (KMS), where generation phases are passed through the knowledge registry until a final knowledge registry is constructed such that it satisfies the requirements of the user (e.g., generating some desired software artifacts).

2.5 Prospective Workflow & Roles

Ideally, the workflow associated with building some product artifact will have each knowledge/product owner (e.g., actual property "*owner*", developers, managers, designers, etc.) work on strictly the components that are related to them, and nothing else. At the "bottom", the final end-user is tasked merely with providing feedback that can improve the quality of the artifact(s), and are the ones that have an issue that can be resolved with some sort of artifact (here, assumed somehow software related). At the "top", product owners will designate a basic set of requirements of the artifact using a coherent *formal language*. Product designers/orchestrators will take the requirements and designate a/communicate it into a coherent *story* for how the requirements may be translated into a final product. The *story* will be built on the well-understood knowledge of various domains encoded by experts of those domains. The product designer is tasked primarily with translation, while the domain developers and product owners are tasked with

encoding knowledge and instances of knowledge, respectively.

Following this ideology, there will be at least 3 key types of roles associated with developing artifacts: the **knowledge encoder**, the **knowledge user**, and the **end-user** of the produced software artifact.

2.5.1 Knowledge Encoder (Domain Expert)

The knowledge encoder should be a master of a particular domain. They are expected to encode the knowledge discussed in their respective domain in such a way that is accessible to those without knowledge of their domain. Additionally, they should encode information about the ways in which the knowledge can be transformed into other forms of knowledge (including that which is interdisciplinary). The knowledge they would be encoding should be as well-known and globally standardized as possible. As discussed in section 2.4, it is likely that the knowledge encoder will focus on writing a series of highly specific domain-specific languages, where the languages may be restricted to as specific as one term or a handful.

2.5.2 Knowledge / Domain User & Orchestrator

The knowledge user/orchestrator should at least be familiar with a particular domain, and have goals in mind for information that they would like to "seed" into their knowledge management system. They should also have a working understanding of what the end-user needs. From this, they should be able to connect the work of the domain expert into "plug-n-play" stories (arguably, compilers for the end-users to use), or be able to encode/reduce friction between knowledge encoded by domain experts.

2.5.3 End-user

The final end-user should find the most delight from this ideology. They are the actual users of the software artifacts, perhaps tweaking the final build of the software artifacts to be accustomed to their workflow. If the tasks assigned to the knowledge encoders and the knowledge users are performed correctly, then the end-user should have strong confidence in the artifacts as they were built with strict adherence to the knowledge captured at *every step of the way*. As such, one should confidently expect the final software artifacts to be completely devoid of unexpected things (including errors, unconformities to specifications, etc.).

Figure out where this below paragraph should belong

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 and that the knowledge is exposed in a usable way (e.g., sufficiently through some sort of API). In other words, all knowledge encodings should create a 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 [2], Agda [8], etc.) — preferring ones with a type system based on formal type theories for their feature richness.

2.6 Feasibility

In order for us to discuss feasibility of this idealized prospective workflow, we must discuss the depth & breadth of knowledge we need to make this feasible. Depth of knowledge refers to the vertical knowledge understood about a specific fragment of knowledge, and it's preciseness. For example, we may have a low-depth of knowledge & claim that English sentences are a sequence of characters, or we might have a slightly "deeper" depth/understanding of sentences by describing them as a language that follows a specific syntax rule set and using a specific set of words. Breadth of knowledge is the horizontal domain of knowledge, it is the various kinds of knowledge we have in a wide variety of subjects and domains.

In low-depth areas, we may observe that this ideology is very practical, and heavily used. Widely used Content Management Systems (CMSs), such as WordPress [9] and Drupal [10], and web frameworks, such as Django [11] and Laravel [12], are arguably also following similar ideals as this ideology. They all typically provide a basic understanding of "users" of a hosted website, facilities to write HTML content in one way or another, plugins, and more. While some might be, these listed above are not specific to one specific use-case. They are very versatile and highly reusable for a wide variety of use-cases because they ship with low but sufficient depth of knowledge (though they might call it "features"). Out of the box, these web technologies listed come with simple, common, functionality (features) and powerful extensibility through either plugins or through software extension and usage. With the basic tooling provided, users are able to rapidly deploy websites with content. Through extending the website's knowledge-base (e.g., plugins or software extension), they are able to obtain a wider breadth and deeper depth to the knowledge contained within them. Through this, the end-users are able to encode increasingly complex and different kinds of data into the systems to ultimately obtain increasingly specialized websites, such as technical

blogs, eCommerce websites, online accounting software, online discussion forums, and more. In these technologies, knowledge depth typically remains "shallow", but through increasing breadth of knowledge, increasingly interesting websites may be created. The mechanized generation-related components of the ideology is also fairly shallow in this area, but, still, highly feasible.

Realistically, almost any area of generation can be a "low-depth"/breadth area too

Add to high knowledge density examples: https://en.wikipedia.org/wiki/Algebraic_modeling_language

Add to high knowledge density examples: https://github.com/McMasterU/HashedExpression

In areas of high knowledge density, as long as developers have infinite patience and can invest infinite time into transcribing knowledge and information into the software, anything is possible, and this ideology is very practical. Unfortunately, that situation is not quite realistic. As such, we should restrict our scope in high knowledge density areas to only those "well-understood" [7]. . In projects with high knowledge density, it's crucial to have clear and concise knowledge fragment encodings that allow users to directly interact with the decomposed simplicity associated with each "transformation" operation. Thankfully, since these domains are also typically codified (or easily codifiable), this is doable. Through capturing many series of incomplex transformations of knowledge into other forms, we are able to sequence and compose them until an ultimate large, normally complex and complicated, transformation is formed. Finally, through creating a directed discussion of ideas, we can form a stable knowledge-base from which we can draw information. One large feasible goal from it is to draw out usable software artifacts from it by poking and prodding in all areas needed until we can deterministically form them, as our needs demand.

Define well-understood

The essence of this ideology lies in naturally obtaining mechanization tech-

niques through formalizing *everything*. The ideology forces us to question modern software development practices: cognitive stress and normal errors aside, if a developer doesn't *truly understand* the knowledge they are encoding in a software product, then it should be normal to expect logical issues and an "imperfect" program. Taking cognitive stress into consideration, there should be considerably less as knowledge only need be transcribed once, and re-used infinitely. Through mechanization, cognitive stress of re-writing knowledge is alleviated afterwards for all proceeding instances. A formalized-knowledge-first approach to software development should highlight areas of issue (poor understanding), never produce bugs, and create software with the same quality as the encoded knowledge.

The ideology also relates to the question of "which programming language do I pick for my project?". This is a moot question under this ideology. There is no one (1) single language that we even use to describe everything, so how can we possibly write all kinds of software for all kinds of knowledge? In a sense, the ideology "leans into it" by saying you shouldn't be limited to choosing just one (1).

The ideology demands that we unify modern domain knowledge. Almost naturally, we obtain mechanization as a side effect. In other words, it demands that we connect all of our compilers into one single coherent mega-compiler system.

One of the large issues with modern software development is a disconnect in knowledge, which is why this paradigm is the natural answer. The ideology is, exactly, "unify all your knowledge." Everything gained by unifying our knowledge is a natural side effect, which we happen to care about.

Through describing their requirements coherently (e.g., some language), completely non-technical product owners may, and will, still be key figures in the production of the product.

For good artifacts to be produced, the task of each product owner in creating coherent stories is critical, or else their biases will show up in the artifacts.

As the stress load/burden becomes shared under this paradigm, the sum of the parts should be less than the whole. In other words, the cumulative stress of associated with creating the whole is greater than the approximate sum of each individual's stress associated with focusing on their respective domain.

Chapter 3

Drasil

Can I edit my own quote? Dr. Smith previously had some notes on this.

> "Drasil is a framework for generating families of software artifacts from a coherent knowledge base, following its mantra, "Generate All The Things!". Drasil uses a series of variably sized Domain-Specific Languages (DSLs) to describe various fragments of knowledge that domain experts and users alike may use to piece together fragments of knowledge into a coherent "story". Through forming some coherent "story" in a domain captured by Drasil, a representational software artifact may be generated. Drasil currently focuses on Scientific Computing Software (SCS), following Smith and Lai's Software Requirements Specification (SRS) template as described in [1]. Behind the scenes of the SRS, a mathematical language is used to describe various theories, and have representational software constructed via compiling to Generic Object-Oriented Language (GOOL) [13]. Through encoding knowledge in Drasil, an increase in productivity (and maintainability) in building reliable and traceable software artifacts is observed [14], specifically in SCS [15]. Drasil's source code (Haskell), case studies, and documentation studies can be found on its website[1]." [16]

[1] https://jacquescarette.github.io/Drasil/

3.1 An Exploration

Cite LSS, or remove this bit of information if there's nothing to cite.

Originally known as Literate Scientific Software (LSS), Drasil is an exploration of this ideology described in Chapter 2. Drasil's largest domain of knowledge covered originates from LSS: scientific computing software. Dr. Jacques Carette[1] and Dr. Spencer Smith[2] are the principal investigators of Drasil. Drasil is deeply embedded in Haskell [2], relying on Stack [17], and compiling against GHC 8.8.4 [18]. Haskell is the language of choice for various reasons, but the most important reasons are regarding its paradigm: purely functional, with immutable data and a strong, sound type system based on Hindley-Milner type system. This provides a sound system developers may use to classify, create, and work with knowledge.

Cite Hindley-Milner type theory.

I should note that when I refer to "Haskell," I'm referring to Haskell 2010 specification and the GHC compiler version 8.

Cite "Haskell 2010" specification being based on Hindley-Milner type theory.

Drasil is currently capable of generating usable software through compiling to Generic Object-Oriented Language (GOOL), which is capable of producing Java, C++, Python, C# [19], and Swift (not discussed in MacLachlan's Master's thesis, but created by him as well, and available similarly). Drasil contains renderers for HTML, Makefile, basic Markdown (enough for README), GraphViz DOT (graph description language) [20], plaintext documents, LaTeX, and TeX. Drasil's source code is publicly available on GitHub[3], and Drasil's documentation (user-facing[4], and internal[5]) is available on the Drasil project homepage[6]. Drasils public wiki is hosted on the same GitHub repository[7], containing information on potential future Drasil projects, Drasil-related papers, a developer workspace configuration

[1] https://www.cas.mcmaster.ca/~carette/
[2] https://www.cas.mcmaster.ca/~smiths/
[3] https://github.com/JacquesCarette/Drasil
[4] https://jacquescarette.github.io/Drasil/docs/index.html
[5] https://jacquescarette.github.io/Drasil/docs/full/index.html
[6] https://jacquescarette.github.io/Drasil/
[7] https://github.com/JacquesCarette/Drasil/wiki

and "quick start" guide[1], and a guide for building your own project with Drasil[2].

3.2 Methodology

Drasils development is strongly influenced by its case studies, which focus on building Scientific Computing Software (SCS) based on solving undergraduate-level physics problems, and using a formalized SCS Software Requirements Specification (SRS) template [1]. A "control" program and SRS are manually created and used as a baseline target for the final artifact. Development is focused on creating a system of justification for each component of the baseline program, following a "bottom-up" agile development approach. Drasil has been constructed following the demands and requirements of eight (8) case studies:

Add citations to each row of the below:

Table 3.1: Drasil Case Studies

Case Study	Focus
Glass Breaking (*GlassBR*)	Predicting likelihood of a glass slab resisting a specified blast.
Projectile (*Projectile*)	Determining if a launched projectile hits a target, assuming no flight collisions.
Single Pendulum (*SglPendulum*)	Observing the motion of a single pendulum.
Double Pendulum (*DblPendulum*)	Observing the motion of a double pendulum.
Game Physics (*GamePhysics*)	Modelling of an open source 2D rigid body physics library used for games.

[1] https://github.com/JacquesCarette/Drasil/wiki/New-Workspace-Setup
[2] https://github.com/JacquesCarette/Drasil/wiki/Creating-Your-Project-in-Drasil

Heat Transfer Coefficients Between Fuel And Cladding In Fuel Rods (*HGHC*)	Examining the heat transfer coefficients related to clad.
Proportional Derivative Controller (*PDController*)	Examining the output of a "Power Plant" (Process Variable) over time.
Solar Water Heating System (*SWHS*)	Modelling of a solar water heating system with phase change material, predicting temperatures and change in heat energy of water and the PCM over time.
Solar Water Heating System Without PCM (*NoPCM*)	Modelling of a solar water heating system without phase change material, predicting temperatures and change in heat energy of water and the PCM over time.
Slope Stability Analysis Program (*SSP*)	Assessment of the safety of a slope (composed of rock and soil) subject to gravity, identifying the surface most likely to experience slip and an index of its relative stability (factor of safety).

3.2.1 Knowledge Dissection and Capture via "Bottom-Up" Gathering

Once baseline/target artifacts are constructed, the essence of the artifacts are dissected and broken into various *fragments* of knowledge. The fragments of knowledge are continuously broken up and explored/understood (via capture through

creating knowledge encodings in Haskell) until a sufficient holistic understanding is formed, generalized, and proven to be capable of re-generating the baseline artifacts. Hopefully, at the end, the baseline artifacts will have their flaws highlighted and resolved in the produced artifacts.

In the end of the bottom-up gathering approach, there will need to be knowledge encodings for, at least:

1. The artifact knowledge encodings — Java, JSON, Python, HTML, SVG, etc. all use a different language.

2. The meta-level knowledge encodings — information lost in the implementations of specific programs needs to be regained so that we can reconstruct it in the original (and other) languages, where possible.

To some degree, the artifact-knowledge encodings may be thought of as a "phenomenon" to Drasil. They may become part of the meta-level knowledge encodings as well, but they are also generally where the side effects of Drasil appear, and file artifacts created.

3.2.2 Overview

The most immediate fragments of relevant knowledge are those that discuss the produced artifacts. Naturally, Drasil is limited to software-based side effects, but focused on producing *files* on a computer.

The case studies are focused on constructing scientific software that follow a precise SRS [1]

Discuss Dr. Smiths template a bit more, I need to explain the sections quickly, give an example, and explain that it dissects SCS into well-understood components, which should provide sufficient information for a developer to build a representational software

. Therefore, all artifacts are produced with the goal of providing a user with a (i) software requirements specification, and (ii) a usable software that adheres to the software requirements specification.

The encodings of the desired artifact files are next most important. For example, an SRS artifact is typically a file that an end-user will be viewing (read-only). Thus, a PDF is appropriate, but also inappropriate due to its complexity, and, hence, we look to constructing SRS documents using a series of files built with a textual markup language, and designating that a secondary compiler should be used to compile the files into a PDF. Naturally, LaTeX, TeX, and HTML are the immediately desired textual markup languages for their usability in scientific scenes, portability, large user-base, and free nature (as opposed to proprietary languages). For a software artifact, there are likely a few relevant fragments, but those most immediately important are the language (programming or otherwise) of the software artifacts and the method to use and compile the software artifact. Generating a Java program will require for Drasil to have a working encoding of Java programs, and a means of directing users on how to use a residual Java program.

Once satisfactory encodings of file types are formed, we must look to the purpose of the files and justify their existence. While a JSON file is strictly for data serialization and deserialization, a Java file is either the "entry" to a program (via it's `main` method) or a file that might be some useful to, and/or used in, a Java program. Therefore, we must create a system of reasoning for why files belong where they do (this might be a part of the language encoding itself as well). For executable software artifacts, the desire is to have an end-user *execute* the final software artifacts produced, and thus we must gain information on what the user expects, and desires. Where feasible, also creating a conceptual understanding how a program meets their expectations and desires. With each of Drasil's case studies, a series of *inputs* are provided to an executable program, and the program

is expected to produce *outputs* that somehow make use of the inputs. Drasils case studies form a system of relations for each study that relates the inputs to the outputs.

As Drasil is focused on forming scientific software that adheres to a specific SRS template [1], the "system of relations" is generally a language of forming a sequence of mathematical calculations. Knowledge is continuously captured until a sufficient fundamental knowledge-base is formed such that a user may generate families of software problems using any relevant domain knowledge. In Drasils case studies, the knowledge capture is heavily influenced by the SRS template [1].

This approach requires that there is sufficient implicit and explicit knowledge around the SRS document such that a user/developer/machine may follow it, without any external assistance (e.g., to clear up ambiguities). This ultimately relies on developing a stable framework for collecting a collection of "cookie-cutter" pieces of knowledge, and interacting with them. By breaking down knowledge into small units of knowledge, the end-result development cycle should appear similar to using a projectional editor with appropriate IDE tooling.

Add an example of an SRS document, and explain how there is sufficient information in it such that we can use it to generate representational software that solves the relevant problems, with little or no manual/extra information (excluding design choices of the desired software artifacts).

Figure 3.1: Drasils Logo

Table 3.2: Drasil Logo Personification

Component	**Conceptual Counterpart/Personification**
Roots	The roots are where the information of the seed influences the earth, and makes it comfortable for the tree to grow tall and firm. Information influences and encourages re-evaluation and structural change of the ground.

Ground / Foundation	The most important component, it is where the tree stands tall, and all knowledge relies on. It contains the definitions of the encodings, and is required to be strong or else a seed will be insufficient, irrelevant of how much topsoil is provided. The ground is irreplaceable, and difficult to "fake".
Seed	The initial bundle of information, from which everything originates and derives from. You provide the bare minimum information to describe your problem, and use nutrients and care to carefully grow the seed into something else.
Nutrients (topsoil and sunlight)	Encouraged growth through hinting/providing extra information. This is where you configure growth and encourage further growth externally, artificially.
Trunk	The initial display of growth of the tree, building a wide knowledge-base. Sometimes requires maintenance (trimming, or, extra information/nutrients) to grow further and become a strong basis for the crown.

Crown + Fruits	The fruits of your labour, standing on the shoulder of giants, where the final product (software artifacts) are realized.

3.3 Architecture

Drasil relies on a domain expert encoding knowledge as a deeply embedded Domain-Specific Language (DSL) or record in Haskell. Instances of the knowledge also appear in the Haskell source code. When knowledge is encoded in Drasil, we refer to it as a *Chunk* or *knowledge fragment* (they will be used interchangeably for the purposes of this thesis). The chunks relevant Haskell-level type is its classifier, and each chunk must have a Unique Identifier (UID) so that one might be able to refer to a specific occurrence of knowledge. The primitive data held within the chunks rely on Haskells primitives, ADTs, and GADTs. Each chunk is a Haskell record, consisting of other encoded data, including instances of Domain-Specific Languages (DSLs)s encoded in Drasil. As chunk types are created, connected, composed, referenced, and intertwined, cookie-cutter principles stories become possible, similar to projectional-style editing. Chunks are funneled into a *Chunk Database* (in Haskell, a `ChunkDB`).

Source Code 3.1: Original Chunk Database (ChunkDB)[1]

```
-- | Our chunk databases. Should contain all the maps we will need.
data ChunkDB = CDB { symbolTable :: SymbolMap
                   , termTable :: TermMap
                   , defTable  :: ConceptMap
                   , _unitTable :: UnitMap
```

[1]https://github.com/JacquesCarette/Drasil/blob/9c26b43d3e30c3f618e534a3f176a5152729af74/code/drasil-database/Database/Drasil/ChunkDB.hs#L130-L144

```
  , _traceTable :: TraceMap
  , _refbyTable :: RefbyMap
  , _dataDefnTable  :: DatadefnMap
  , _insmodelTable   :: InsModelMap
  , _gendefTable   :: GendefMap
  , _theoryModelTable :: TheoryModelMap
  , _conceptinsTable :: ConceptInstanceMap
  , _sectionTable :: SectionMap
  , _labelledcontentTable :: LabelledContentMap
  } --TODO: Expand and add more databases
```

A `ChunkDB` is currently limited to the above listed chunks (Source Code 3.1), but one should assume that more data types are also *chunks* because this is merely a temporary restriction at the moment. The types of each record item is approximately a map from a `UID` to an instance of a chunk (Source Code A.4). Relating back to the ideology (as discussed in chapter 2), in Drasil, the `ChunkDB` is where the knowledge is collected, placed, and grown (generated). *Transformers* are used to operate on the `Chunk`s in the `ChunkDB`. The transformers take in some knowledge, optional *refinement* information, and convert it into some other knowledge fragment. As expected, these transformations are "against the grain" of the "bottom-up" information gathering. Areas of precise and highly specific fragments of knowledge have their key components used in generating something in another language (this typically involves some sort of information loss/strip in order to restrict the knowledge for a specific purpose). The produced fragments are directed by the developer. By composing a series of miniature, "bite-sized", transformations, we are able to create large and highly complex (but well-thought-out and understood).

Provide a text-based example.

The largest, most prominent transformer used in Drasil is that of the *SmithEtAl* knowledge transformer. It is currently represented using the entry point to Haskell programs, the `main :: IO ()` function is representative of the transformation capabilities of our understanding of our case studies into sufficient SRS documents, and ready-to-use software.

I'm naming this, which I really shouldn't be doing, but I need a name to refer to the SRS+Code generator we

Source Code 3.2: GlassBR "main" Function[1]

```
main :: IO()
main = do
  setLocaleEncoding utf8
  gen (DocSpec (docChoices SRS [HTML, TeX]) "GlassBR_SRS") srs
↪  printSetting
  genCode choices code
  genDot fullSI
  genLog fullSI printSetting
```

Source Code 3.2 displays an example, from the GlassBR case study, of what a transformer looks like in Drasil. This transformer takes knowledge collected, configures a transformation task for the desired outputs, as prescribed by an orchestrating developer, and performs an `IO ()` effect that dumps an SRS and a software artifact to the host computer, in the local working directory. While not seemingly coupled, they indeed are. The code generator focuses on understanding fragments/ideas relevant to generating the SRS, with more stringent rules than the SRS generator to create a coherent software artifact. The data collected in the `ChunkDB`s are collected at Drasil Haskell-level compile-time, instead of the Drasil run-time, because the chunks are captured within Haskell source code instead of an external resource. In other words, the data is embedded in Haskell itself.

Code snippet: "genSRS"

Code snippet: "genCode"

Code snippet: "PrintingInformation", "SystemInformation", & "Choices"

Discuss how PrintingInformation, SystemInformation, and Choices are used to configure a SmithEtAl transformer run.

The SRS and software artifacts are optional features. They are also not always guaranteed to function merely because they were executed. It is up to the

[1] https://github.com/JacquesCarette/Drasil/blob/9f435cbb9bcda515d765c551606d1f4e3d9fc101/code/drasil-example/glassbr/app/Main.hs#L8-L14

knowledge transcribers to ensure that the body of knowledge recorded is logically consistent and well-understood to Drasil. In particular, the SmithEtAl transformer requires that the recorded *theories*/models are consistent, and that, together, they are able to produce a meta-level holistic single theory that connects a list of inputs to a list of outputs of a desired calculation. The calculations should also be reasonably convertible in some programming language if we decide to use the calculations for code generation as well.

Coloured diagram showing flow of information and how general information flow works.

Table 3.3: Drasil Case Studies Artifacts Generated

Case Study	SRS	C/C++	Java	C#	Python	Swift
GlassBR	✓	✓	✓	✓	✓	✓
Projectile	✓	✓	✓	✓	✓	✓
SglPendulum	✓					
DblPendulum	✓					
GamePhysics	✓					
HGHC	✓					
PDController	✓				✓	
SWHS	✓					
NoPCM	✓	✓	✓	✓	✓	
SSP	✓					

Table 3.3 shows a quick overview of what final artifacts Drasil is capable of generating for each case study. The SRS is the simplest to generate because it is not tested for logical consistency and usability by a software developer. As such, the SRS is built for each for case study. Some case studies (GamePhysics, HGHC, and SSP) are still actively being developed, but are left incomplete at the time of writing. The SRS is currently generated in both LaTeX and HTML flavours, with

the LaTeX variant having supplementary build information for building to a single PDF file, and the HTML variant accessible from a web browser compliant with HTML version 5 standards. The GlassBR, Projectile, PDController, and NoPCM case studies each are capable of generating representational software. NoPCM is usable in all languages supported by GOOL except for Swift due to the lack of a Drasil-supported ODE solving library for the Swift GOOL renderer. PDController was built outside of the normal means of Drasils case studies development, being built by a student at McMaster University for a class taken. Code generation for PDController is not impossible, it just requires more investigation for the needs of the case study. However, both the issues related to NoPCM and PDController are outside the scope of this work.

Cite Naveens work.

The code generator works by following the requirements as set in the SRS document, applying transformations until a sufficient knowledge-base can be formed such that a residual OO program can be generated via GOOL. The SRS document (and relevant encodings of the knowledge in the SRS documents) outlines the input and output variables of a SCS. The Data Definitions, Instance Models, and symbols are all used to form a "calculation path" that relates the inputs to the outputs of the desired software artifact. If such calculation path cannot be found, then no program will be generated, and an error will be displayed to the user at Drasils runtime (e.g., it is not checked at Drasil compile-time). Each executable software artifact produced is coupled with a means (currently, a generated Makefile) to run and, where required, build the software artifact. Additionally, for supported languages, Doxygen [21] configurations may be built.

Link to an example in the Appendix?

Link to an example in the Appendix?

The remaining case studies that do not generate code are still left. Issues related to their encodings of mathematical knowledge plague their usability in the generation process.

Chapter 4

Framing Theories

In working to understand some phenomenon, we often look to the boundaries of our understandings of the phenomenon — the open-ended questions and gaps in our knowledge. In any domain of knowledge (such as mathematics, philosophy, physics, chemistry, biology, computer science, or any other studied subject), we often think about various phenomenon in logical terms of theories and axioms, where the theories and axioms are written using a formal DSL tailored to the field of knowledge. As this work pertains to Drasil, one language of interest to this research is that of the mathematical language. A mathematical language is heavily used in science, conveying important theories, axioms, corollaries, amongst other things. As mathematicians transferring knowledge to one another, we often use a written form of knowledge, though unlikely, potentially, with a precise structure to our transcription of the knowledge, but still, the structure is textual. We might break up the transcription of the knowledge into its logical constituents. The components of a theory may be a name, a natural language description, a derivation, some information regarding its origin (e.g., a reference), and a formalization of the theories important information in a precise language. Of course, for a mathematical theory, a mathematical language would be used. Describing a programming language artifact, we might prefer to call our theories by a series of other names

(e.g., functions/methods, constants, modules, packages, libraries, APIs, ASTs).

In Drasil, both of these languages are of great importance. The mathematical field of knowledge is directly used in at least two domains currently captured: the scientific theories, and GOOL [13]. Meanwhile, the programming language theory (artifacts) is only directly used in GOOL, and its various renderers. The SmithEtAl SRS [1] template generator uses knowledge transcribed as scientific requirements to automatically generate a series of representational software programs. Thankfully, Smith and Lai [1] break down theories relevant to SCS into at least four (4) kinds for their SRS template:

Insider knowledge: the 3 "models" are actively being re-thought, and *are* being renamed to just "theories".

Cite "Theory Presentations."

1. **Theory Models**: A "theory presentation" : an abstract theory, highly general and unspecific, with room for refinement. Typically, these are names coupled with information contained as axioms and data that make up a coherent story, but that are too vague to be immediately usable by the developers other than for refining into Instance Models.

Discuss what refinement means here.

2. **General Definitions**: A partially refined theory, typically bearing a stronger relation to the context of the SRS than the Theory Models. However, still, these are generally theoretical and abstract, much like Theory Models.

3. **Instance Models**: A fully refined theory, derived from the Theory Models and General Definitions, and directly usable by implementors of the SRS.

4. **Data Definitions**: Symbols that are defined with basic expressions of other symbols and inputs. Typically, these are definitions that are "given", and used for making inputs easier to work with for the instance models.

Example of a theory/instance model here.

The Instance Models and Data Definitions are to be directly used in the solving algorithm for the whole system, while the General Definitions and Theory Models provide justification for the existence of the Instance Models, and the Data Definitions are provided and relatively unimportant/uncritical to the system.

Since the SmithEtAl generator focuses itself on knowledge captured in the form of the SRS template, Drasil requires a comprehensive understanding of both fields in order to generate representational software for the SRS. The focus of the scientific requirements documents are to transcribe the relevant knowledge of a SCS system for end-user developers to satisfy when developing SCS, and as developed by a domain/scientific expert. For example, is used to fully explain such that a software developer can construct a program based on it. Of course, if the SRS document is to be truly sufficient for a software developer to unambiguously create a representational software artifact, then the software developer (who potentially knows nothing about the domain discussed in the SRS document) will need to be able to credibly transcribe everything into their program, matching the requirements as set by the SRS document. Drasil is an apparatus for testing our understanding of these scenarios, rather than having a read-only "view" of the scientists/domain experts knowledge available to the software developer, the knowledge itself is available, in its most raw encoded form.

SRS document example

something

Drasil relies on searching for a calculation path that relates the inputs and outputs designated in an SRS document. The relations are based on the grounded theories (*Instance Models*) designated in the SRS. The relations themselves are currently required to be of the form $x = f(a, b, c, \dots)$, with some exceptions made for ODEs. ODEs are being actively developed to remove the manually written exceptions made for them (outside the scope of this work).

Instance Models, General Definitions, and Data Definitions each rely on a *Relation Concept* (`RelationConcept`): a notable and named mathematical relation with a description and abbreviated name. A `RelationConcept` is modelled as a

`Relation` coupled with a `ConceptChunk` (Source Code A.3, a named *thing* with a name, abbreviation, and natural language description):

Source Code 4.1: Original RelationConcept Definition[1]

```
data RelationConcept = RC { _conc :: ConceptChunk
                          , _rel :: Relation
                          }
```

The `Relations` (Source Code A.1) in the `RelationConcept` are, fundamentally, just an instance of a universal mathematical language which conveys their information, but given a type synonym to indicate that the instance should be a mathematical relation.

Theory Models as well rely on this same mathematical language used in `RelationConcepts` to describe their own theories as well. The encoding is slightly better as it exposes more of the expressions, but, still, the expressions are left problematic.

4.1 A Universal Math Language

Source Code 4.2: Original Expression language[2]

```
-- | Drasil Expressions
data Expr where
  Dbl      :: Double -> Expr
  Int      :: Integer -> Expr
  Str      :: String -> Expr
  Perc     :: Integer -> Integer -> Expr
  AssocA   :: ArithOper -> [Expr] -> Expr
  AssocB   :: BoolOper  -> [Expr] -> Expr
  -- | Derivative, syntax is:
```

[1] https://github.com/JacquesCarette/Drasil/blob/9c26b43d3e30c3f618e534a3f176a5152729af74/code/drasil-lang/Language/Drasil/Chunk/Relation.hs#L14-L16

[2] https://github.com/JacquesCarette/Drasil/blob/9c26b43d3e30c3f618e534a3f176a5152729af74/code/drasil-lang/Language/Drasil/Expr.hs#L40-L82

```
  --   Type (Partial or total) -> principal part of change -> with
↪  respect to
  --   For example: Deriv Part y x1 would be (dy/dx1)
  Deriv    :: DerivType -> Expr -> UID -> Expr
  -- | C stands for "Chunk", for referring to a chunk in an
↪  expression.
  --   Implicitly assumes has a symbol.
  C        :: UID -> Expr
  -- | F(x) is (FCall F [x] []) or similar.
  --   FCall accepts a list of params and a list of named params.
  --   F(x,y) would be (FCall F [x,y]) or sim.
  --   F(x,n=y) would be (FCall F [x] [(n,y)]).
  FCall    :: UID -> [Expr] -> [(UID, Expr)] -> Expr
  -- | Actor creation given UID and parameters
  New      :: UID -> [Expr] -> [(UID, Expr)] -> Expr
  -- | Message an actor:
  --   1st UID is the actor,
  --   2nd UID is the method
  Message  :: UID -> UID -> [Expr] -> [(UID, Expr)] -> Expr
  -- | Access a field of an actor:
  --   1st UID is the actor,
  --   2nd UID is the field
  Field :: UID -> UID -> Expr
  -- | For multi-case expressions, each pair represents one case
  Case     :: Completeness -> [(Expr,Relation)] -> Expr
  Matrix   :: [[Expr]] -> Expr
  UnaryOp  :: UFunc -> Expr -> Expr
  BinaryOp :: BinOp -> Expr -> Expr -> Expr
  -- | Operators are generalized arithmetic operators over a
↪  |DomainDesc|
  --   of an |Expr|.  Could be called |BigOp|.
  --   ex: Summation is represented via |Add| over a discrete
↪  domain
  Operator :: ArithOper -> DomainDesc Expr Expr -> Expr -> Expr
  -- | element of
  IsIn     :: Expr -> Space -> Expr
  -- | a different kind of 'element of'
  RealI    :: UID -> RealInterval Expr Expr -> Expr
```

I should probably discuss the ADT portion a bit more.

`Expr` (defined above, in Source Code 4.2 with an ADT) represents the hypothetical mathematical language used to discuss the mathematics relevant to common SCS, specifically, at least to under/graduate-level physics problems. The language contains the commonly found operations in a well-understood physics textbook

(here, with a focus on graduate-level scientific problems). The mathematical language is universal, covering a wide range of knowledge, including facilities for creating commonly used primitive data types, operations, and functions in mathematics, physics, and computer science and programming languages. For example, is transcribed in Drasil as follows:

An expression.

Transcription of the above expression.

Refà new appendix entry for new smart constructors.

This transcription relies on smart constructors, such as . The smart constructors used are all specialized to `Expr` and perform simplifications along the way (such as folding $1 \cdot x$ into x).

4.2 A Universal Theory Description Language

Refthe definitions of DDs, IMs, GDs, and TMs

As Drasils theories rely heavily on `Relations` (and `RelationConcepts`) (), we may observe that the theories are an accurate reflection of writing out mathematics involved as one might write them down on paper: exactly as they please. In modelling any problem, one will, of course, model the work of their pencil and paper. This is exactly what occurs here. Theories here are shallow representations of knowledge, that mimic your pencil and papers work to a clueless reader.

4.3 Usage: Converting to Software and Specifications

Drasil relies on converting the various kinds of theories described in the SRS template [1], essentially encoded in Drasil as `RelationConcepts`, into representational code.

Example of an IMs conversion into Java code.

Unfortunately, issues occur when attempting to convert the knowledge contained in `RelationConcepts`. Namely,

I should probably rewrite the below point form notes as paragraphs as well, even though I originally intended to have it as a numbered list.

Source Code 4.3: Original relToQD[1]

```
-- Converts a chunk with a defining relation to a QDefinition
relToQD :: ExprRelat c => ChunkDB -> c -> QDefinition
relToQD sm r = convertRel sm (r ^. relat)

-- Converts an Expr representing a definition (i.e. an equality
↪  where the left
-- side is just a variable) to a QDefinition.
convertRel :: ChunkDB -> Expr -> QDefinition
convertRel sm (BinaryOp Eq (C x) r) = ec (symbResolve sm x) r
convertRel _ _ = error "Conversion failed"
```

1. The transformation of `RelationConcept` → *GOOL* is not *total* (i.e., not all terms of the `Expr` language can be mapped into a representational GOOL term).

    - Not all terms of the `Expr` language have a definite value.
    - Some terms in `Expr` require extra information before they can be converted into code. At times, this information is a conscious choice that the developer should be making instead of imposed on them.

2. It is very easy to write "difficult/impossible to interpret" expressions. For example, it is possible to create expressions for which aren't directly calculable (i.e., things that require an extra paper and pencil/mental mathematics before performing), either without extra surrounding information or simply impossible.

    - When writing with pencil and paper, we usually write with extra context (generally more information that needs to be read to fully under-

[1] https://github.com/JacquesCarette/Drasil/blob/9c26b43d3e30c3f618e534a3f176a5152729af74/code/drasil-code/Language/Drasil/CodeSpec.hs#L112-L120

stand some expression), assuming the reader understands that context. We might also infer information about the model. Unfortunately, mechanizing inference, without any sort of context or other knowledge, is difficult, artificial learning is a whole field of study of its own, and we're not interested in it here! As such, we reverse the relationship of the inference by having knowledge container expose everything on its own. We additionally expect knowledge as a requisite to working with the model.

- Transformation requires a comprehensive understanding of the inputs to outputs translation, but much of the input knowledge requires complex analysis that would only appear in the transformer, discarding its usability elsewhere — information loss.
- An `Expr` alone is a weak conveyor of the inner knowledge of theories, similar to normal pencil-and-paper mathematical expressions, without extra information, the expression alone may be ineffectual or nearly unusable in code generation.
- It is important that each knowledge encoding in Drasil exposes as much information as reasonably possible (and useful). We want to expose the "specifications" of each piece of knowledge that we are encoding so that transformers and generators may appropriately make use of contained knowledge.

3. Drasil is limited to using theories with expressions written in a very precise form: $x = f(a, b, c, \dots)$

Discuss `QDefinition`s.

- The $=$ sign is being overloaded here to mean definition, when it is supposed to mean equality. While $y = x$ might conventionally be seen as "y is equal to x", we might want, in our model, for it to be understood as "x is defined by y" but displayed differently.

- Any other form of theories are unusable. Equational constraints at the very least should be immediately usable in creating assertions in the generated code.
- The conversion relied on the `relToQD` hack.
- The above is what leads to the brittle untotal conversion of theories to code.

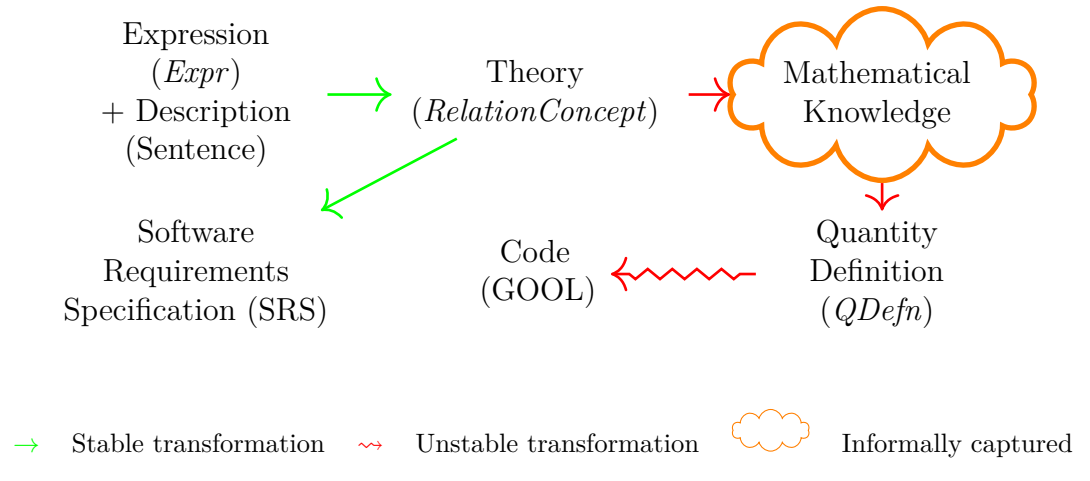
4. Existing transformation of `RelationConcept`s into GOOL relies on unstable transformation caused by need to write in a precise digestible form, with no static checks done at compile-time.

Mention ODE -> code generation.

Add code snippet of original relToQD and discuss how it wasn't a sustainable solution for the long-term. relToQD was exactly the implicit knowledge I mentioned, it assumes, without statically checking consistency, that the Relations/Exprs provided as input for code generation were all of the form: $y = f(\dots)$. See https://github.com/JacquesCarette/Drasil/issues/628 for more information about state of

Therefore, mathematical knowledge flow is unstable in Drasil, as shown in Figure 4.1. The focal issue with the existing way theories are encoded is that the *theories (`RelationConcept`s) do not contain the meaningful usable mathematical knowledge at all.* The mathematical knowledge is implicitly held within the Haskell-based functions. In order to proceed, this mathematical knowledge must be reconciled and merged with the `RelationConcept`s in some capacity.

Figure 4.1: Mathematical Knowledge Flow



Do I need to cite this figure as originating from my poster? It was edited a bit for this thesis.

4.4 Reconciling Mathematical Knowledge

Fundamentally, the issue lies in that the information exposed for Drasil to make use of is too "shallow". The raw expressions are great for viewing and human-guided inference by experts, but not an inexperienced person, or for a computer to systematically use to generate things. At the moment, the available tools' type signatures would appear as something too amazing, so much so in fact, that one would (and should) question its accuracy. A hypothetical type signature, `RelationConcept -> Code`, is quite far fetched. Not all expressions that can possibly be contained in a `RelationConcept` are usable in code generation.

Note somewhere in the thesis above that when we discuss "code", we really mean OO programs and snippets

4.5 Language Division

GOOL currently bears the burden of creating a family of software artifacts through describing a single GOOL program. As we are specifically interested in generating families of SCS through strong knowledge capture, this work partially bears the burden of testing to ensure that the conversion of the knowledge contained in the SmithEtAl template [1] is stable and reliable. At the moment, it is possible to describe mathematical expressions that are completely unusable in GOOL because GOOLs primary targets are languages that rely on expressions with, at least, definite values. Of course, `Expr` already contains terms that are without definite values. For example, `Expr` (Source Code 4.2) discusses derivations, integrations, spaces as primitive/literal values (largely symbolic, and unavailable in OO languages), definition statements, equivalence, and infinite series. Realistically, this information already *exists* in Drasil, but it is not strongly enforced at

I guess it's less-so about definite value, moreso about lack of representation in

Drasils Haskell-level compile-time. In other words, the information is not *exposed* to Haskells type system, but to Drasils runtime.

In order to ensure that developers can only write "usable" grounded theories ("InstanceModels") for the code generator to use, we have chosen to expose this information to the compiler. Specifically, we have chosen to expose it via language division and type information.

Figure 4.2: Mathematical Language Division

$$\texttt{Expr} \Rightarrow \texttt{Expr} \cup \texttt{ModelExpr} \cup \texttt{CodeExpr}$$

Re: TTF: What Haskell/GHC language extensions did we need to use?

List notable terms moved, Deriv, Continuous vs Discrete ranges, Space, etc

i.e., ...

`Expr` has its indefinite values split off into `ModelExpr` (Source Code A.7), a deeply embedded universal mathematical language. As we desire for it to be equally usable as `Expr`, it is, essentially, the original `Expr` without the language terms related to "code". In order to alleviate the stress involved with writing out the same expression in two (2) different languages, we use a Typed Tagless Final (TTF) encoding of the two (2) languages (Source Code A.6 and Source Code A.8). The TTF encoding allows us to seamlessly write expressions in either, or both, languages at the same time. `ModelExpr`s TTF encoding strictly contains the terms unique to `ModelExpr`. As such, the TTF encoding won't allow terms from `ModelExpr` to be interpreted into `Expr` unless they have a definite value because it will be impossible to describe the terms unique to `ModelExpr` in terms of the `Expr` language. However, it is possible (and is done normally) to convert `Expr`s into `ModelExpr`s for usage in generating the SRS documents, which primarily expect `ModelExpr`s for transcribing mathematical expressions. Furthermore, with `ModelExpr`, we may use instances of `Express` (Source Code 4.4) to define how various chunks can be described visually, using this mathematical modelling language.

Source Code 4.4: Current Express Typeclass[1]

```haskell
-- | Data that can be expressed using 'ModelExpr'.
class Express c where
  express :: c -> ModelExpr
```

Continuing, we've noticed that there are residual terms left in `Expr` that don't quite relate to mathematics, but to "code" specifically (here, "code" meaning the OO "code"). As such, we continue the division by moving those less related into their own language, specifically tailored to "code" expressions: `CodeExpr` ().

ref Current CodeExpr Haskell

Finally, we end up with 3 languages, as shown in Figure 4.2: `Expr` (Source Code A.5), `ModelExpr` (Source Code A.7), and `CodeExpr` (). Each language has its own specific domain, and, although there is some overlap between each, we are able to enforce weaker rules on each formation through their TTF instances if needed.

ref Current CodeExpr Haskell

Relating back to Figure 4.1, we may observe that there is a transformation from *Theory* (realistically, these expression languages discussed) to *Mathematical Knowledge*. This may seem peculiar because one might expect the *theory* to be precisely the *mathematical knowledge*. Presently, the mathematical knowledge is implicitly built into transformers that work with `RelationConcept`s (theories). The explicit information is *lost* in both the Haskell-level transformation function that makes generation possible and the external knowledge used to create the actual expression itself. To resolve this, we need to reconcile *theories* with *mathematical knowledge*, strengthening the *depth of knowledge* contained in a theory. Should this occur, we should observe Figure 4.1 having the *Theory* and *Mathematical Knowledge* nodes merged, and have the expressions understood to only be one of many possible "views" of higher-level usable knowledge. In other words,

[1] https://github.com/JacquesCarette/Drasil/blob/dc3674274edb00b1ae0d63e04ba03729e1dbc6f9/code/drasil-lang/lib/Language/Drasil/ExprClasses.hs#L9-L11

the expressions would not be used to transfer knowledge any longer, but they might remain as one component of it. Through resolving these issues, we will have deeper knowledge available at Drasils compile-time, and we will be able to better understand which theories are usable in code generation, and which aren't. Additionally, we will be able to better handle more *kinds* of theories without needing to create complex traversal and analysis algorithms to recognize when certain kinds of theories were transcribed in the expressions.

4.6 "Classify All The Theories"

Issues occurring due to weak knowledge capture may be resolved through strong knowledge capture. Beginning with the existing case studies of Drasil, we will attempt to classify our existing knowledge better. We aim to make `RelationConcept` a "view" of other more information-dense encodings. In other words, we replace `Expr` as a knowledge container, and restrict its usage to strictly "mathematical expressions", as opposed to "expressions" and information about models/theories. One notable change is that we will require the new theory knowledge containers to be able to fully re-create the original shallow/raw `Expr`s as a property of the new theory encodings. The once meta-level knowledge of the theories, lost in the Haskell implementation, becomes exposed and understood to Drasil. Ultimately, this is done through replacing `RelationConcept` usage with `ModelKind`, an aggregation of existing Drasil-related knowledge of mathematical theories. `ModelKind` is defined using a GADT, with one (1) type parameter. The type parameter is currently used to determine whether the model is "fully refined"/"grounded" or not, and, hence, usable in code generation. Source Code 4.5 displays the creation of `ModelKind` and `ModelKinds`. Please note that this aggregation is based purely on the existing model examples in the existing Drasil case studies, and the existing models are *incomplete* in the larger scope.

What must each kind of model provide for them to be candidates for a ModelKind type? One: they must be able to re-create the original Expr they had been encoded as, in the new ModelExpr language, via their "Express" instance.

Source Code 4.5: ModelKinds[1]

```haskell
-- | Models can be of different kinds:
--
--     * 'NewDEModel's represent differential equations as
↪  'DifferentialModel's
--     * 'DEModel's represent differential equations as
↪  'RelationConcept's
--     * 'EquationalConstraint's represent invariants that will
↪  hold in a system of equations.
--     * 'EquationalModel's represent quantities that are
↪  calculated via a single definition/'QDefinition'.
--     * 'EquationalRealm's represent MultiDefns; quantities that
↪  may be calculated using any one of many 'DefiningExpr's (e.g.,
↪  'x = A = ... = Z')
--     * 'FunctionalModel's represent quantity-resulting function
↪  definitions.
--     * 'OthModel's are placeholders for models. No new
↪  'OthModel's should be created, they should be using one of the
↪  other kinds.
data ModelKinds e where
  NewDEModel            :: DifferentialModel -> ModelKinds e
  DEModel               :: RelationConcept   -> ModelKinds e --
↪  TODO: Split into ModelKinds Expr and ModelKinds ModelExpr
↪  resulting variants. The Expr variant should carry enough
↪  information that it can be solved properly.
  EquationalConstraints :: ConstraintSet e   -> ModelKinds e
  EquationalModel       :: QDefinition e     -> ModelKinds e
  EquationalRealm       :: MultiDefn e       -> ModelKinds e
  OthModel              :: RelationConcept   -> ModelKinds e --
↪  TODO: Remove (after having removed all instances of it).


-- | 'ModelKinds' carrier, used to carry commonly overwritten
↪  information from the IMs/TMs/GDs.
```

[1] https://github.com/JacquesCarette/Drasil/blob/dc3674274edb00b1ae0d63e04ba03729e1dbc6f9/code/drasil-theory/lib/Theory/Drasil/ModelKinds.hs#L25-L47

```
data ModelKind e = MK {
  _mk     :: ModelKinds e,
  _mkUID  :: UID,
  _mkTerm :: NP
}
```

First TODO in Source Code 4.5 is seemingly wrong, I need to clarify that with NewDEModel vs DEModel

4.6.1 Quantity Definitions

Source Code 4.6: Current QDefinition Encoding[1]

```
data QDefinition e where
  QD :: DefinedQuantityDict -> [UID] -> e -> QDefinition e
```

Assume $y = x$ is transcribed as a `RelationConcept`: while $y = x$ might conventionally be seen as "y is equal to x", we might want, in our model, for it to be understood as "x is defined by y" but displayed differently. Here, = is overloaded as "definition", instead of what = was defined as in `Expr`, as an "equality" operator. To resolve this overloading and weak knowledge capture of definitions, we create `EquationalModel`s: theories that contain information about definitions of symbols, built using a `QDefinition` (Source Code 4.6). If an `EquationalModel` deals with theoretical symbols and is defined using either a `ModelExpr` or an `Expr`, it may be used in Theory Models and General Definitions. If an `EquationalModel` is defined using an `Expr` and deals with only the non-abstract symbols, then the `EquationalModel` is usable for code generation. At the moment, there is no information attached to symbols yet regarding whether they are abstract or instanced, so that portion of the rule is not enforced.

Is there a name for this?

Future Work: make typed distinction between abstract and instanced symbols

[1] https://github.com/JacquesCarette/Drasil/blob/ab9e091dabd81685ddef86b0d218582c9f75cb20/code/drasil-lang/lib/Language/Drasil/Chunk/Eq.hs#L34-L35

Example of an EquationalModel/QDefinition in Haskell code, the SRS, and the generated code.

4.6.2 Constraints

Source Code 4.7: Current ConstraintSet Definition[1]

```haskell
-- | 'ConstraintSet's are sets of invariants that always hold for
↪  underlying domains.
data ConstraintSet e = CL {
    _con  :: ConceptChunk,
    _invs :: NE.NonEmpty e
}
```

Describe what a ConstraintSet actually is.

Theories that define expressions that constrain models in some way are defined using `EquationalConstraints`, which use `ConstraintSet`s under the hood (Source Code 4.7). At the moment, these are not used in code generation, and are pending design for usage in code generation as the translation from them into software is unclear and may be interpreted differently by readers. This model can, and should, eventually also be usable in code generation once there is interest in creating runtime assertions of symbols.

Example of an EquationalConstraints/ConstraintSet in Haskell code, and the SRS.

4.6.3 Definition Realms

Source Code 4.8: Definition Possibility[2]

[1] https://github.com/JacquesCarette/Drasil/blob/9f435cbb9bcda515d765c551606d1f4e3d9fc101/code/drasil-theory/lib/Theory/Drasil/ConstraintSet.hs#L15-L19

```haskell
-- | 'DefiningExpr' are the data that make up a (quantity)
 ↪  definition, namely
--   the description, the defining (rhs) expression and the context
 ↪  domain(s).
--   These are meant to be 'alternate' but equivalent definitions
 ↪  for a single concept.
data DefiningExpr e = DefiningExpr {
  _deUid  :: UID,       -- ^ UID
  _cd     :: [UID],     -- ^ Concept domain
  _rvDesc :: Sentence, -- ^ Defining description/statement
  _expr   :: e          -- ^ Defining expression
}
```

Source Code 4.9: "MultiDefinitions" (MultiDefn) Definition[1]

```haskell
-- | 'MultiDefn's are QDefinition factories, used for showing one
 ↪  or more ways we
--   can define a QDefinition.
data MultiDefn e = MultiDefn {
    _rUid  :: UID,                          -- ^ UID
    _qd    :: QuantityDict,                 -- ^ Underlying quantity
 ↪  it defines
    _rDesc :: Sentence,                     -- ^ Defining
 ↪  description/statement
    _rvs   :: NE.NonEmpty (DefiningExpr e) -- ^ All
 ↪  possible/omitted ways we can define the related quantity
           -- TODO: Why is this above constraint redundant
 ↪  according to the smart constructors?
}
```

The last TODO note in this MultiDefn code snippet is 'bad'.

The Theory Models and General Definition Models are unrefined, and may contain multiple ways for a particular theoretical symbol to be defined. This is captured in Drasil by `EquationalRealm`s: modelled after realms. This model kind is intended specifically for retaining information about conscious choices made along

Cite realms? If so, from Dr. Carette and Yasmines paper?

[2]https://github.com/JacquesCarette/Drasil/blob/dc3674274edb00b1ae0d63e04ba03729e1dbc6f9/code/drasil-theory/lib/Theory/Drasil/MultiDefn.hs#L19-L27

[1]https://github.com/JacquesCarette/Drasil/blob/dc3674274edb00b1ae0d63e04ba03729e1dbc6f9/code/drasil-theory/lib/Theory/Drasil/MultiDefn.hs#L35-L43

the way to create Instance Models. An `EquationalRealm` is based on a `MultiDefn` (Source Code 4.9), and is intended for forming `QDefinition`s through choosing definitions (encoded as `DefiningExpr`s, Source Code 4.8), through refinements, from the `MultiDefn`s.

Example of an EquationalRealm/MultiDefn in Haskell code, and the SRS.

4.6.4 Differential Equations

`DEModel` is the simple `RelationConcept`-style capture of differential equation-related theories, and only exists as a placeholder until all differential equation models are converted into one of the other `ModelKinds`. For related solving of differential equations, `DEModel` implicitly relies on a developer writing a related `ODEInfo` packet which the code generator can use to solve the system. Actively being reconstructed in Drasil, `NewDEModel` is the replacement for `DEModel`, aiming to expose more information about well-understood differential equations and related areas. `NewDEModel` will eventually be renamed to `DEModel` once all existing `DEModel` models have been upgraded.

ref Current ODEInfo

Cite Dongs work?

Example of a DEModel/NewDEModel and a RelationConcept in Haskell code, the SRS, and the related ODEInfo and code.

4.6.5 Leftovers

Instead of writing anything here, I'm going to re-evaluate the existing leftover models and see if they can be replaced with any of the existing or if they need a new one.

- `OthModel` ...

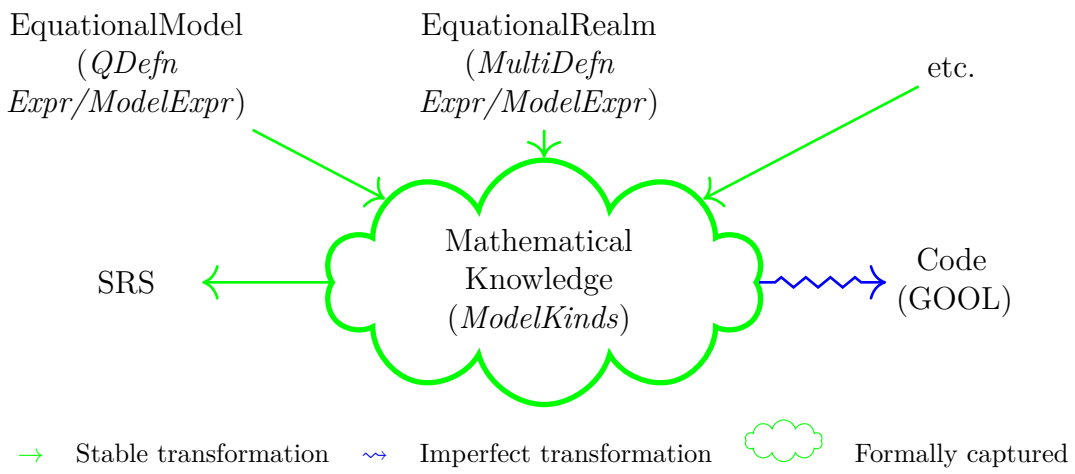
4.7 Theories Undiscussed

`ModelKinds` is an enumeration of the currently handled model types. Each "model"

is meant to expose information to the Haskell compiler and for other fragments of knowledge to make use of. `ModelKinds` is obviously an incomplete enumeration, and will grow as the need for more kinds of models arises.

Through incorporating `ModelKinds` and reaping its benefits, we obtain, approximately, the flow of mathematical knowledge as shown below, in Figure 4.3.

Figure 4.3: Mathematical Knowledge Flow, with Formal Capture



Do I need to cite this figure as originating from my poster? It was edited a bit for this thesis.

Comparing Figure 4.3 with Figure 4.1, we observe a decrease in the scary orange colouring! The focal difference in the two figures (and the difference between the work pre-existing and current) lies in the understanding of a theory. Previously, the "theory" was captured by the `RelationConcept`s themselves, captured fully by a single `Expr` (the original universal version). The mathematical knowledge of the theory was never explicitly captured, and usage of theories was unsustainable due to reliance on brittle transformations (`relToQD`, Source Code 4.3) and implicit expectations of the written transcriptions. Now, the theories and the mathematical knowledge are unified with information exposed both by dissection of components as record entries, and by creating type information for classifying theories and relevant information. Notably, from the new encodings of the theories, we are still

able to recreate the original `Expr`s (or, now `ModelExpr`s) that once represented them.

Side-by-side figure comparison of Figure 4.3 and Figure 4.1

What do we learn from doing this conversion, about the greater part about encoding knowledge? ...

From this, we obtain a new general rule for future knowledge dissection/capture/encoding: when extracting information about the formation of certain language snippets and classifying that contextual information, we must ensure that we have obtained a generic means (algorithm) for reproducing the originating term from an instance of the new contextual information. In other words, there should be a generic method to reproduce the original information from the new information — this is finding areas where we can add to the generation possibilities of Drasil.

4.8 Residual Issues

While the information is now encoded, it is not without its own issues:

1. As `ModelKinds` is written using a GADT, it is difficult to extend its functionality (the "expression problem"). A work in progress re-design of `ModelKinds`, relying on GHCs `ConstraintKinds` [22] language extension, uses Haskell-level typeclasses to describe what a model satisfying the needs of being a "model kind" must provide in order to be treated as a "model kind". These typeclass instances may be thought of as proofs that satisfy the requirements.

Cite the original expression problem paper.

2. The type parameter in `ModelKinds` is peculiar. It is used to indicate usability of a certain model in code generation through either having an `Expr` or a `ModelExpr`. A `ModelKind Expr` indicates that a model is usable in code generation, while a `ModelKind ModelExpr` indicates that the model is not to be used in code generation.

3. While we've gained information about the reason what the expressions described, the expression language, as a whole, allows for poorly typed expressions to be formed. We lack enforcement regarding the validity of `Expr` as "expressions".

However, these first two (2) issues are relatively unimportant for now. (1) has a proof of concept solution (as discussed above), and (2) is a matter of the Haskell implementation we choose. (3) is the next area of focus in regard to expressions.

Chapter 5

Expression Formation Rules

Rewrite the point form notes in the Typing section.

5.1 Background: Problem

- Writing invalid expressions is possible.
  - On paper, writing invalid expressions is as easy as making a typo, but complete gibberish can also be written. We rely on manually checking expressions to ensure that they are "correct". As the number of expressions grows, the cost of manually checking grows rapidly, and changes result in costly setbacks. Imagine systems with 10, 100, and 1000+ expressions, the cost grows rapidly.
  - With computers, we can systematically check the validity of expressions by imposing various kinds of restrictions.
- Mentally tracking expression creations to ensure they follow the implicit rules of the expression language is too difficult, and leads to mental strain.
- Compiling to "lower languages" requires special type checking before compiling to them. For example, the Swift code generator has to ensure that

there are no ambiguously typed numerals as the types of numerics are not overloaded in Swift.

- Dynamically checking for invalid expression states is possible, but difficult and would result in increasingly difficult term tracking as terms in the expression language grow/are added.
- In general, being able to express invalid expressions causes large burden and mental overhead.

5.2 Requirements & properties of a good solution

- Invalid expressions should not be representable in the various expression languages (i.e., the expression types should strictly indicate valid expression constructions), without loss of generality.
- Invalid expression formation attempts should be statically found and reported by the compiler, at compile-time. This will move the previously runtime errors to compile-time.
- Invalid expression cases should not need to be considered when working (e.g., case-ing) with expressions.
- "Safety = Preservation + Progress" ([23], Ch.6)

5.3 Solution

- Use TTF encodings of the smart constructors to lessen the cognitive load of handling at least 3 different expression languages.
- Statically type all 3 variants of Expr through GADTs.

5.3.1 Syntax

Current

An idealized version of the current syntax.

Type	τ	::=	`Integer`	$\mathbb{Z}$	Integer numbers
		|	`Real`	$\mathbb{R}$	Real numbers
		|	`String`	$String$	Text
		|	`Bool`	$\mathbb{B}$	Truth values (tru
		|	`Vector(`τ`, `n`)`	$[\tau]_n$	Vectors (single el
		|	`Tuple(`$\tau_1 ... \tau_n$`)`	$\tau_1 \times \tau_2 \times ... \times \tau_n$	Alternative vecto
Literal	l	::=	`Integer[`n`]`	n	Integer number
		|	`Real[`r`]`	r	Real number
		|	`String[`s`]`	"s"	Text
		|	`Bool[`b`]`	b	Boolean value
		|	`Vector(`$l_1 ... l_n$`)`	$< l_1, ..., l_n >$	Vectors*
		|	`Tuple(`$l_1 ... l_n$`)`	$(l_1, ..., l_n)$	Tuples*
UnaryOp	$\ominus$	::=	`Not`	$\neg\_$	Logical negation
		|	`Neg`	$-\_$	Numeric negation
		|	`...`		omitted for brevity
BinaryOp	$\oplus$	::=	`Sub`	$\_ - \_$	Subtraction
		|	`Pow`	$\_^-$	Powers
		|	`...`		omitted for brevity
AssocBinOp	$\otimes$	::=	`Add`	$\_ + \_$	Addition
		|	`Mul`	$\_ \times \_$	Multiplication

		| ...		omitted for brevity
UID	u	::=	`UID(s)`	`UID "s"`	UIDs
Expr	e	::=	`Literal(`l`)`	l	Literal values
		|	`Vector(`$e_1 ... e_n$`)`	$< e_1, ..., e_n >$	Vectors
		|	`Var(`u`)`	u	Variable (Quanti
		|	`FuncCall(`$f, e_1 ... e_n$`)`	$f(e_1 ... e_n)$	"Complete" funct
		|	`UnaryOp(`$\ominus$`,`e`)`	$\ominus\ e$	Unary operations
		|	`BinaryOp(`$\oplus$`,`e_1`,`e_2`)`	$e_1 \oplus e_2$	Binary operation
		|	`AssocOp(`$\otimes$`,` $e_1 ... e_n$`)`	$e_1 \otimes ... \otimes e_n$	Associative binar
		|	`Case(`$e_{1c} e_{1e} ... e_{nc} e_{ne}$`)`	$if\ e_{1c}\ then\ e_{2e}\ elif\ e_{2c}\ ...$	If-then-else-if-the
		|	`BigAsBinOp(`$\otimes, e_1, e_2$`)`	$\bigotimes_{i=e_1}^{e_2} i$	Apply a "big" op
		|	`IsInRlItrvl(`u, e_1, e_2`)`	$u \in [e_1, e_2]$	Variable in range

$*$: does not currently appear in the code at the moment, but would be needed/desired

5.3.2 Typing Rules

Literal

1. Integers:

$$\frac{i : \texttt{Integer}}{Integer[i] : \texttt{Literal Integer}} \tag{5.1}$$

2. Strings (Text):

$$\frac{s : \texttt{String}}{Str[s] : \texttt{Literal String}} \tag{5.2}$$

3. Real numbers:

$$\frac{d : \texttt{Double}}{Dbl[d] : \texttt{Literal Real}} \tag{5.3}$$

4. Whole numbered reals ($\mathbb{Z} \subset \mathbb{R}$):

$$\frac{d : \texttt{Integer}}{ExactDbl[d] : \texttt{Literal Real}} \tag{5.4}$$

5. Percentages:

$$\frac{n : \texttt{Integer} \quad d : \texttt{Integer}}{Perc[n, d] : \texttt{Literal Real}} \tag{5.5}$$

Miscellaneous

Sorts Legend	**Numerics**(T)	: any numeric type
	NumericsWithNegation(T)	: any signed numeric type

These might need to be replaced with variants for Reals/Integers

Vectors

As of right now, Drasil/GOOL only supports lists and arrays as "code types", which would be the representations used for representing "vectors" in Drasil.

Do we want to have the length of our vectors as a type argument?

For now, the below type rules define vectors with Haskell lists. We can choose to create our own type with the length of the vector as a parameter – likely going "too far into Haskell".

Functions

Presently, functions are defined through "QDefinitions", where a list of UIDs used in an expression are marked as the parameters of the function. Function "calls"/applications are captured in "Expr" (the expression language) by providing a list of input expressions and a list of named inputs (expressions) – $f(x, y, z, a = b)$.

define criteria for what a well-formed expression language should provide

Quantities discussion – remaining untyped

A few solutions:

1. Leave expressions in general untyped in Haskell, and rely on calculating the "space" of an expression dynamically to ensure that expressions are well-formed. If runtime (Drasil's compiling-knowledge-time) type analysis is ever needed, this will prove much easier to use in general.

2. Push the typing rules into Haskell via Generalized Algebraic Data Types (GADTs). Here, a larger question appears regarding functions – how should we handle function creation, application, and typing?

   (a) Currying and applying arguments (allowing partial function applications): This would work well if we only generated functional languages, but it might prove problematic for GOOL if expressions are left with partial function applications.

Type Rules

1. Completeness:

$$\frac{}{Complete[] : \texttt{Completeness}} \tag{5.6}$$

$$\frac{}{Incomplete[] : \texttt{Completeness}} \tag{5.7}$$

2. AssocOp:

   (a) Numerics:

$$\frac{x : \texttt{Numerics(T)}}{Add[] : \texttt{AssocOp x}} \tag{5.8}$$

$$\frac{x : \texttt{Numerics(T)}}{Mul[] : \texttt{AssocOp x}} \tag{5.9}$$

   (b) Bool:

$$\frac{}{And[] : \texttt{AssocOp Bool}} \tag{5.10}$$

$$\frac{}{Or[] : \texttt{AssocOp Bool}} \tag{5.11}$$

3. UnaryOp:

   (a) Numerics:

$$\frac{x : \texttt{NumericsWithNegation(T)}}{Neg[] : \texttt{UnaryOp x x}} \tag{5.12}$$

$$\frac{x : \texttt{NumericsWithNegation(T)}}{Abs[] : \texttt{UnaryOp x x}} \tag{5.13}$$

$$\frac{x : \texttt{Numerics(T)}}{Exp[] : \texttt{UnaryOp x Real}} \tag{5.14}$$

For Log, Ln, Sin, Cos, Tan, Sec, Csc, Cot, Arcsin, Arccos, Arctan, and Sqrt, please use the following template, replacing "$\$TRG$" with the desired operator:

$$\frac{}{\$TRG[] : \texttt{UnaryOp Real Real}} \tag{5.15}$$

$$\frac{}{RtoI[] : \texttt{UnaryOp Real Integer}} \tag{5.16}$$

$$\frac{}{ItoR[] : \texttt{UnaryOp Integer Real}} \tag{5.17}$$

$$\frac{}{Floor[] : \texttt{UnaryOp Real Integer}} \tag{5.18}$$

$$\frac{}{Ceil[] : \texttt{UnaryOp Real Integer}} \tag{5.19}$$

$$\frac{}{Round[] : \texttt{UnaryOp Real Integer}} \tag{5.20}$$

$$\frac{}{Trunc[] : \texttt{UnaryOp Real Integer}} \tag{5.21}$$

(b) Vectors:

$$\frac{x : \texttt{NumericsWithNegation(T)}}{NegV[] : \texttt{UnaryOp [x] [x]}} \tag{5.22}$$

$$\frac{x : \texttt{Numerics(T)}}{Norm[] : \texttt{UnaryOp [x] Real}} \tag{5.23}$$

$$\frac{\texttt{x} : \tau}{Dim[] : \texttt{UnaryOp [x] Integer}} \tag{5.24}$$

(c) Booleans:

$$\frac{}{Not[] : \texttt{UnaryOp Bool Bool}} \tag{5.25}$$

4. BinaryOp:

(a) Arithmetic:

$$\frac{}{FracI[] : \texttt{BinaryOp Integer Integer Integer}} \tag{5.26}$$

$$\frac{}{FracR[] : \texttt{BinaryOp Real Real Real}} \tag{5.27}$$

modulo, remainder, etc.

(b) Bool:

$$\frac{}{Impl[] : \texttt{BinaryOp Bool Bool Bool}} \tag{5.28}$$

$$\frac{}{Iff[] : \texttt{BinaryOp Bool Bool Bool}} \tag{5.29}$$

(c) Equality:

$$\frac{\texttt{x} : \tau}{Eq[] : \texttt{BinaryOp x x Bool}} \tag{5.30}$$

$$\frac{\texttt{x} : \tau}{NEq[] : \texttt{BinaryOp x x Bool}} \tag{5.31}$$

(d) Ordering:

$$\frac{x : \texttt{Numerics(T)}}{Lt[] : \texttt{BinaryOp x x Bool}} \tag{5.32}$$

$$\frac{x : \texttt{Numerics(T)}}{Gt[] : \texttt{BinaryOp x x Bool}} \tag{5.33}$$

$$\frac{x : \texttt{Numerics(T)}}{LEq[] : \texttt{BinaryOp x x Bool}} \tag{5.34}$$

$$\frac{x : \texttt{Numerics(T)}}{GEq[] : \texttt{BinaryOp x x Bool}} \tag{5.35}$$

(e) Indexing:

$$\frac{\texttt{x} : \tau}{Index[] : \texttt{BinaryOp [x] Integer x}} \tag{5.36}$$

(f) Vectors:

$$\frac{x : \texttt{Numerics(T)}}{Cross[] : \texttt{BinaryOp [x] [x] [x]}} \tag{5.37}$$

$$\frac{x : \texttt{Numerics(T)}}{Dot[] : \texttt{BinaryOp [x] [x] x}} \tag{5.38}$$

$$\frac{x : \texttt{Numerics(T)}}{Scale[] : \texttt{BinaryOp [x] x [x]}} \tag{5.39}$$

5. RTopology:

$$\frac{}{Discrete[] : \texttt{RTopology}} \tag{5.40}$$

$$\frac{}{Continuous[] : \texttt{RTopology}} \tag{5.41}$$

6. DomainDesc:

oddity: topology appears as a constructor arg and signature arg but can desync – can we just remove the constructor arg?

$$\frac{top : \tau_1 \quad bot : \tau_2 \quad s : \texttt{Symbol} \quad rtop : \texttt{RTopology}}{BoundedDD[s, rtop, top, bot] : \texttt{DomainDesc Discrete } \tau_1 \ \tau_2} \tag{5.42}$$

$$\frac{\texttt{topT} : \tau \quad \texttt{botT} : \tau \quad s : \texttt{Symbol} \quad rtop : \texttt{RTopology}}{AllDD[s, rtop] : \texttt{DomainDesc Continuous topT botT}} \tag{5.43}$$

7. Inclusive:

$$\frac{}{Inc[] : \texttt{Inclusive}} \tag{5.44}$$

$$\frac{}{Exc[] : \texttt{Inclusive}} \tag{5.45}$$

8. RealInterval:

$$\frac{\texttt{a} : \tau \quad \texttt{b} : \tau \quad top : \texttt{(Inclusive, a)} \quad bot : \texttt{(Inclusive, b)}}{Bounded[top, bot] : \texttt{RealInterval a b}} \tag{5.46}$$

$$\frac{\texttt{a} : \tau \quad \texttt{b} : \tau \quad top : \texttt{(Inclusive, a)}}{UpTo[top] : \texttt{RealInterval a b}} \tag{5.47}$$

$$\frac{\texttt{a} : \tau \quad \texttt{b} : \tau \quad bot : \texttt{(Inclusive, b)}}{UpFrom[bot] : \texttt{RealInterval a b}} \tag{5.48}$$

Expr

1. Literals:

$$\frac{\texttt{x} : \tau \quad l : \texttt{Literal x}}{Lit[l] : \texttt{Expr x}} \tag{5.49}$$

2. Associative Operations:

$$\frac{\texttt{x} : \tau \quad op : \texttt{AssocOp x} \quad args : \texttt{[Expr x]}}{Assoc[op, args] : \texttt{Expr x}} \tag{5.50}$$

3. Symbols:

$$\frac{\texttt{x} : \tau \quad u : \texttt{UID}}{C[u] : \texttt{Expr x}} \tag{5.51}$$

4. Function Call: Addressed in "misc" section.

5. Case:

$$\frac{\texttt{x} : \tau \quad c : \texttt{Completeness} \quad ces : \texttt{[(Expr Bool, Expr x)]}}{Case[c, ces] : \texttt{Expr x}} \tag{5.52}$$

6. Matrices:

$$\frac{\texttt{x} : \tau \quad es : \texttt{[[Expr x]]}}{Matrix[es] : \texttt{Expr x}} \tag{5.53}$$

7. Unary Operations:

$$\frac{\texttt{x} : \tau \quad \texttt{y} : \tau \quad op : \texttt{UnaryOp x y} \quad e : \texttt{Expr x}}{Unary[op, e] : \texttt{Expr y}} \tag{5.54}$$

8. Binary Operations:

$$\frac{\texttt{x} : \tau \quad \texttt{y} : \tau \quad \texttt{z} : \tau \quad op : \texttt{BinaryOp x y z} \quad l : \texttt{Expr x} \quad r : \texttt{Expr y}}{Binary[op, l, r] : \texttt{Expr z}} \tag{5.55}$$

9. "Big" Operations:

$$\frac{\texttt{x} : \tau \quad op : \texttt{AssocOp x} \quad dom : \texttt{DomainDesc Discrete (Expr x) (Expr x)}}{BigOp[op, dom] : \texttt{Expr x}} \tag{5.56}$$

10. "Is in interval" operator:

$$\frac{\texttt{x} : \tau \quad u : \texttt{UID} \quad itvl : \texttt{RealInterval (Expr x) (Expr x)}}{RealI[u, itvl] : \texttt{Expr x}} \tag{5.57}$$

ModelExpr

1. $$\frac{B \quad C}{A}$$

CodeExpr

1. $$\frac{B \quad C}{A}$$

Continued problems? Expressions don't expose enough information to be used in softifact generation

Chapter 6

"Store All The Things"

Rewrite the point form notes in Storing Chunks chapter.

- `UID`s
- ChunkDB: Multiple maps from `UID`s to single types
- Problems occur:
  - `UID` collisions
  - Difficult to ascertain what a specific chunk type is from a `UID`
  - "ChunkDB" is not a stable core across Drasil-like projects (ones that thrive on the same "knowledge-based programming" ideology). `ChunkDB`s are essentially the "scope" of a system.
- Solution:
  - Merge the maps!
  - The key would be the same; a `UID`.
  - The value type?
  - An existentially quantified `Data.Typeable.Typeable`!
  - e.g., `data Chunk = forall a. Typeable => Chunk a`.

- But wait! We're missing a few things from chunks:
  - What knowledge does the chunk rely on already having been "registered" in the database and ready?
  - They should have `UID`s; where's our guarantee?
  - Debugging will be difficult; need an interface to dump all information of a chunk quickly.
- Ok, revise: `data Chunk = forall a. (Typeable a, HasUID a, HasChunkRefs a, Du`
- Ok, much better now.
- Or is it? Still many problems!
  - How do we explain "Data.Typeable"?
  - And "HasUID"?
  - And "HasChunkRefs"?
  - And "Dumpable"?
- Well, at the very least, now we're able to merge the "chunk" maps and fix many of the pre-existing problems (we're almost there!). However, now we're relying too much on Haskell. How do we explain those parts?
- Also, what are `UID`s really? Do their information carry any real information? *Rigid designators.*

6.1 Future Work

6.1.1 Encodings

- With the above new definition of "chunks", they still remain a very vague idea, and still *deeply embedded* (a place to recognize an encoding might be

appropriate!) in Haskell.

- What are the kinds of chunks that can exist? What can be in a chunk, and what are we missing from the existing list of chunks?
- The problem with that is that we lose a lot of information by writing Haskell, and leaving the knowledge in the form of Haskell.
- We need to de-embed all chunks so that we can obtain a tangible understanding of them.
- Through de-embedding the chunks, we will also be forced to de-embed everything with it. This is including the ways in which we transform and generate "new"-ish knowledge (not necessarily new types/kinds of knowledge, but new instances of types).

1. What is a "chunk"?
    - A "chunk instance" is a single *term* of a language.
    - A "chunk type" is a language itself.
2. What is a "transformer"?
    - A "transformer" is a conversion of a term written in a language into another term, potentially in another language.
    - Transformers rely on a well-understood dissection of knowledge (contained in a chunk type/language) in order transform it (potentially with other terms/information as well) into another term.

Chapter 7

Future Work

Rewrite point form notes in Future Work chapter.

- This document will contain knowledge regarding the Expression language that is shown in Haskell code, but not quite in our encodings. To further improve Drasil, one of the next "obvious" steps is to transcribe the knowledge involved with writing any language down as well. An excruciating amount of knowledge is everywhere.
- The unit and dimension related to numbers is another project on its own. It will need to be added to calculate the units of operations and ensure that representations are appropriate for their (precision vs accuracy as Dr. Smith mentioned).

Chapter 8

Conclusion

Write a conclusion chapter.

Bibliography

[1] W. Spencer Smith and Lei Lai. "A New Requirements Template for Scientific Computing". In: *Proceedings of the First International Workshop on Situational Requirements Engineering Processes – Methods, Techniques and Tools to Support Situation-Specific Requirements Engineering Processes, SREP'05*. In conjunction with 13th IEEE International Requirements Engineering Conference. Paris, France, 2005, pp. 107–121 (cit. on pp. 3, 4, 21, 23, 25, 27, 36, 40, 44).

[2] Simon Marlow et al. *Haskell 2010 Language Report*. https://www.haskell.org/onlinereport/haskell2010/. 2010 (cit. on pp. 4, 16, 22).

[3] Krzysztof Czarnecki. "Overview of Generative Software Development". In: *Unconventional Programming Paradigms*. Ed. by Jean-Pierre Banâtre, Pascal Fradet, Jean-Louis Giavitto, and Olivier Michel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 326–341. ISBN: 978-3-540-31482-0 (cit. on p. 6).

[4] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. "Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages". In: *Journal of Functional Programming* 19.5 (2009), pp. 509–543 (cit. on p. 8).

[5] GitHub ™ and OpenAI ™. *Copilot*. 2021. URL: https://copilot.github.com/ (cit. on p. 10).

[6] Wikipedia. *Vasa (ship) — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Vasa%20(ship)&oldid=1081988142. [Online; accessed 20-April-2022]. 2022 (cit. on p. 11).

[7] Jacques Carette, Spencer Smith, and Jason Balaci. "When Capturing Knowledge Improves Productivity". Submitted Nov 2021 to NIER - New Ideas and Emerging Results (ICSE 2022). 2021. URL: https://github.com/JacquesCarette/Drasil/blob/master/Papers/WellUnderstood/wu.pdf (cit. on pp. 14, 18).

[8] Ulf Norell. "Towards a practical programming language based on dependent type theory". PhD thesis. Chalmers University of Technology and Göteborg University, 2007 (cit. on p. 16).

[9] WordPress Foundation. *WordPress*. URL: https://wordpress.org/ (cit. on p. 17).

[10] Dries Buytaert. *Drupal*. 2001. URL: https://www.drupal.org/ (cit. on p. 17).

[11] Django Software Foundation. *Django*. 2005. URL: https://www.djangoproject.com/ (cit. on p. 17).

[12] Taylor Otwell. *Laravel*. 2011. URL: https://laravel.com/ (cit. on p. 17).

[13] Jacques Carette, Brooks MacLachlan, and W. Spencer Smith. *GOOL: A Generic Object-Oriented Language (extended version)*. 2019. DOI: 10.48550/ARXIV.1911.11824. URL: https://arxiv.org/abs/1911.11824 (cit. on pp. 21, 36).

[14] Daniel Szymczak, W. Spencer Smith, and Jacques Carette. "Position Paper: A Knowledge-Based Approach to Scientific Software Development". In: *Proceedings of SE4Science'16 in conjunction with the International Conference*

on Software Engineering (ICSE). Austin, Texas, United States, 2016-05 (cit. on p. 21).

[15] W. Spencer Smith. "Beyond Software Carpentry". In: *2018 International Workshop on Software Engineering for Science (held in conjunction with ICSE'18)*. 2018, pp. 32–39 (cit. on p. 21).

[16] Jason Balaci. "Capturing Mathematical Knowledge in Drasil: the Case of Theories". In: 2021. Poster presented at McMaster University, Hamilton, Canada (cit. on p. 21).

[17] FP Complete and Stack contributors. *Stack*. 2015. URL: https://docs.haskellstack.org/en/stable/README/ (cit. on p. 22).

[18] The Glasgow Haskell Team. *Glasgow Haskell Compiler (GHC): GHC 8.8.4 User's Guide*. https://downloads.haskell.org/~ghc/8.8.4/docs/html/users_guide/index.html. 2020 (cit. on p. 22).

[19] Brooks MacLachlan. "A Design Language for Scientific Computing Software in Drasil". Thesis. McMaster University, 2020-07 (cit. on p. 22).

[20] Emden Gansner, Eleftherios Koutsofios, Stephen North, and Khoi Vo. "A Technique for Drawing Directed Graphs". In: *Software Engineering, IEEE Transactions on* 19 (1993-04), pp. 214–230. DOI: 10.1109/32.221135 (cit. on p. 22).

[21] Dimitri van Heesch. *Doxygen*. 1997. URL: https://www.doxygen.nl/index.html (cit. on p. 34).

[22] The Glasgow Haskell Team. *Glasgow Haskell Compiler User's Guide*. 2020-07. URL: https://downloads.haskell.org/~ghc/8.8.4/docs/html/users_guide/glasgow_exts.html#the-constraint-kind (cit. on p. 54).

[23] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2016 (cit. on p. 57).

Appendix

Source Code A.1: Relation[1]

```haskell
type Relation = Expr
```

Source Code A.2: Snapshot of a few of Exprs Smart Constructors[2]

```haskell
-- | Smart constructor to apply tan to an expression
tan :: Expr -> Expr
tan = UnaryOp Tan

-- | Smart constructor to apply sec to an expression
sec :: Expr -> Expr
sec = UnaryOp Sec

-- | Smart constructor to apply csc to an expression
csc :: Expr -> Expr
csc = UnaryOp Csc

-- | Smart constructor to apply cot to an expression
cot :: Expr -> Expr
cot = UnaryOp Cot
```

Source Code A.3: Original ConceptChunk[3]

[1] https://github.com/JacquesCarette/Drasil/blob/9c26b43d3e30c3f618e534a3f176a5152729af74/code/drasil-lang/Language/Drasil/Expr.hs#L14

[2] https://github.com/JacquesCarette/Drasil/blob/9c26b43d3e30c3f618e534a3f176a5152729af74/code/drasil-lang/Language/Drasil/Expr/Math.hs

[3] https://github.com/JacquesCarette/Drasil/blob/9c26b43d3e30c3f618e534a3f176a5152729af74/code/drasil-lang/Language/Drasil/Chunk/Concept/Core.hs#L25-L29

```haskell
-- | The ConceptChunk datatype is a Concept
data ConceptChunk = ConDict { _idea :: IdeaDict
                            , _defn' :: Sentence
                            , cdom' :: [UID]
                            }
```

Source Code A.4: Original ChunkDB Type Maps[1]

```haskell
-- The misnomers below are not actually a bad thing, we want to
 ↪  ensure data can't
-- be added to a map if it's not coming from a chunk, and there's
 ↪  no point confusing
-- what the map is for. One is for symbols + their units, and the
 ↪  others are for
-- what they state.
type UMap a = Map.Map UID (a, Int)

-- | A bit of a misnomer as it's really a map of all quantities,
 ↪  for retrieving
-- symbols and their units.
type SymbolMap  = UMap QuantityDict

-- | A map of all concepts, normally used for retrieving
 ↪  definitions.
type ConceptMap = UMap ConceptChunk

-- | A map of all the units used. Should be restricted to base
 ↪  units/synonyms.
type UnitMap = UMap UnitDefn

-- | Again a bit of a misnomer as it's really a map of all
 ↪  NamedIdeas.
-- Until these are built through automated means, there will
-- likely be some 'manual' duplication of terms as this map will
 ↪  contain all
-- quantities, concepts, etc.
type TermMap = UMap IdeaDict
type TraceMap = Map.Map UID [UID]
type RefbyMap = Map.Map UID [UID]
type DatadefnMap = UMap DataDefinition
```

[1] https://github.com/JacquesCarette/Drasil/blob/9c26b43d3e30c3f618e534a3f176a5152729af74/code/drasil-database/Database/Drasil/ChunkDB.hs#L18-L47

```haskell
type InsModelMap = UMap InstanceModel
type GendefMap = UMap GenDefn
type TheoryModelMap = UMap TheoryModel
type ConceptInstanceMap = UMap ConceptInstance
type SectionMap = UMap Section
type LabelledContentMap = UMap LabelledContent
```

Source Code A.5: Current Expression Language[1]

```haskell
-- | Expression language where all terms are supposed to be 'well
 ↪  understood'
--   (i.e., have a definite meaning). Right now, this coincides
 ↪  with
--   "having a definite value", but should not be restricted to
 ↪  that.
data Expr where
  -- | Brings a literal into the expression language.
  Lit :: Literal -> Expr
  -- | Takes an associative arithmetic operator with a list of
 ↪  expressions.
  AssocA   :: AssocArithOper -> [Expr] -> Expr
  -- | Takes an associative boolean operator with a list of
 ↪  expressions.
  AssocB   :: AssocBoolOper  -> [Expr] -> Expr
  -- | C stands for "Chunk", for referring to a chunk in an
 ↪  expression.
  --   Implicitly assumes that the chunk has a symbol.
  C        :: UID -> Expr
  -- | A function call accepts a list of parameters and a list of
 ↪  named parameters.
  --   For example
  --
  --   * F(x) is (FCall F [x] []).
  --   * F(x,y) would be (FCall F [x,y]).
  --   * F(x,n=y) would be (FCall F [x] [(n,y)]).
  FCall    :: UID -> [Expr] -> [(UID, Expr)] -> Expr
  -- | For multi-case expressions, each pair represents one case.
  Case     :: Completeness -> [(Expr, Relation)] -> Expr
  -- | Represents a matrix of expressions.
  Matrix   :: [[Expr]] -> Expr
```

[1] https://github.com/JacquesCarette/Drasil/blob/dc3674274edb00b1ae0d63e04ba03729e1dbc6f9/code/drasil-lang/lib/Language/Drasil/Expr/Lang.hs#L81-L135

```
  -- | Unary operation for most functions (eg. sin, cos, log,
↪  etc.).
  UnaryOp       :: UFunc -> Expr -> Expr
  -- | Unary operation for @Bool -> Bool@ operations.
  UnaryOpB      :: UFuncB -> Expr -> Expr
  -- | Unary operation for @Vector -> Vector@ operations.
  UnaryOpVV     :: UFuncVV -> Expr -> Expr
  -- | Unary operation for @Vector -> Number@ operations.
  UnaryOpVN     :: UFuncVN -> Expr -> Expr
  -- | Binary operator for arithmetic between expressions
↪  (fractional, power, and subtraction).
  ArithBinaryOp :: ArithBinOp -> Expr -> Expr -> Expr
  -- | Binary operator for boolean operators (implies, iff).
  BoolBinaryOp  :: BoolBinOp -> Expr -> Expr -> Expr
  -- | Binary operator for equality between expressions.
  EqBinaryOp    :: EqBinOp -> Expr -> Expr -> Expr
  -- | Binary operator for indexing two expressions.
  LABinaryOp    :: LABinOp -> Expr -> Expr -> Expr
  -- | Binary operator for ordering expressions (less than, greater
↪  than, etc.).
  OrdBinaryOp   :: OrdBinOp -> Expr -> Expr -> Expr
  -- | Binary operator for @Vector x Vector -> Vector@ operations
↪  (cross product).
  VVVBinaryOp   :: VVVBinOp -> Expr -> Expr -> Expr
  -- | Binary operator for @Vector x Vector -> Number@ operations
↪  (dot product).
  VVNBinaryOp   :: VVNBinOp -> Expr -> Expr -> Expr
  -- | Operators are generalized arithmetic operators over a
↪  'DomainDesc'
  --   of an 'Expr'.  Could be called BigOp.
  --   ex: Summation is represented via 'Add' over a discrete
↪  domain.
  Operator :: AssocArithOper -> DiscreteDomainDesc Expr Expr ->
↪  Expr -> Expr
  -- | A different kind of 'IsIn'. A 'UID' is an element of an
↪  interval.
  RealI    :: UID -> RealInterval Expr Expr -> Expr
```

Source Code A.6: Current Expr Constructor Encoding (TTF)[1]

```
class ExprC r where
  infixr 8 $^
```

[1] https://github.com/JacquesCarette/Drasil/blob/ab9e091dabd81685ddef86b0d218582c9f75cb20/code/drasil-lang/lib/Language/Drasil/Expr/Class.hs#L58-L212

```
infixl 7 $/
infixr 4 $=
infixr 9 $&&
infixr 9 $||

lit :: Literal -> r

-- * Binary Operators

($=), ($!=) :: r -> r -> r

-- | Smart constructor for ordering two equations.
($<), ($>), ($<=), ($>=) :: r -> r -> r

-- | Smart constructor for the dot product of two equations.
($.) :: r -> r -> r

-- | Add two expressions (Integers).
addI :: r -> r -> r

-- | Add two expressions (Real numbers).
addRe :: r -> r -> r

-- | Multiply two expressions (Integers).
mulI :: r -> r -> r

-- | Multiply two expressions (Real numbers).
mulRe :: r -> r -> r

($-), ($/), ($^) :: r -> r -> r

($=>), ($<=>) :: r -> r -> r

($&&), ($||) :: r -> r -> r

-- | Smart constructor for taking the absolute value of an
↪  expression.
abs_ :: r -> r

-- | Smart constructor for negating an expression.
neg :: r -> r

-- | Smart constructor to take the log of an expression.
log :: r -> r

-- | Smart constructor to take the ln of an expression.
```

```
ln :: r -> r

-- | Smart constructor to take the square root of an expression.
sqrt :: r -> r

-- | Smart constructor to apply sin to an expression.
sin :: r -> r

-- | Smart constructor to apply cos to an expression.
cos :: r -> r

-- | Smart constructor to apply tan to an expression.
tan :: r -> r

-- | Smart constructor to apply sec to an expression.
sec :: r -> r

-- | Smart constructor to apply csc to an expression.
csc :: r -> r

-- | Smart constructor to apply cot to an expression.
cot :: r -> r

-- | Smart constructor to apply arcsin to an expression.
arcsin :: r -> r

-- | Smart constructor to apply arccos to an expression.
arccos :: r -> r

-- | Smart constructor to apply arctan to an expression.
arctan :: r -> r

-- | Smart constructor for the exponential (base e) function.
exp :: r -> r

-- | Smart constructor for calculating the dimension of a vector.
dim :: r -> r

-- | Smart constructor for calculating the normal form of a
↪  vector.
norm :: r -> r

-- | Smart constructor for negating vectors.
negVec :: r -> r
```

```
  -- | Smart constructor for applying logical negation to an
↪  expression.
  not_ :: r -> r

  -- | Smart constructor for indexing.
  idx :: r -> r -> r

  -- | Smart constructor for the summation, product, and integral
↪  functions over an interval.
  defint, defsum, defprod :: Symbol -> r -> r -> r -> r

  -- | Smart constructor for 'real interval' membership.
  realInterval :: HasUID c => c -> RealInterval r r -> r

  -- | Euclidean function : takes a vector and returns the sqrt of
↪  the sum-of-squares.
  euclidean :: [r] -> r

  -   TODO:  sum' :: (Num a, Foldable t) => t a -> a
  -   TODO:  sum' = foldr1 (+)

  -- | Smart constructor to cross product two expressions.
  cross :: r -> r -> r

  -- | Smart constructor for case statements with a complete set of
↪  cases.
  completeCase :: [(r, r)] -> r

  -- | Smart constructor for case statements with an incomplete set
↪  of cases.
  incompleteCase :: [(r, r)] -> r

  -- | Create a matrix.
  -- TODO: Re-work later.
  matrix :: [[r]] -> r


  -- TODO: The 3 below smart constructors can be re-built above
↪  without needing to be inside of this typeclass definition.


  -- | Create a two-by-two matrix from four given values. For
↪  example:
  --
  -- >>> m2x2 1 2 3 4
  -- [ [1,2],
```

```
  --   [3,4] ]
  m2x2 :: r -> r -> r -> r -> r

  -- | Create a 2D vector (a matrix with two rows, one column).
↪  First argument is placed above the second.
  vec2D :: r -> r -> r

  -- | Creates a diagonal two-by-two matrix. For example:
  --
  -- >>> dgnl2x2 1 2
  -- [ [1, 0],
  --   [0, 2] ]
  dgnl2x2 :: r -> r -> r

  -- Some helper functions to do function application

  -- FIXME: These constructors should check that the UID is
↪  associated with a
  -- chunk that is actually callable.
  -- | Applies a given function with a list of parameters.
  apply :: (HasUID f, HasSymbol f) => f -> [r] -> r

  -- | Similar to 'apply', but takes a relation to apply to
↪  'FCall'.
  applyWithNamedArgs :: (HasUID f, HasSymbol f, HasUID a,
↪  IsArgumentName a) => f
    -> [r] -> [(a, r)] -> r

  -- Note how |sy| 'enforces' having a symbol
  -- | Create an 'Expr' from a 'Symbol'ic Chunk.
  sy :: (HasUID c, HasSymbol c) => c -> r
```

Source Code A.7: Current ModelExpr Language[1]

```
-- | Expression language where all terms are supposed to have a
↪  meaning, but
--   that meaning may not be that of a definite value. For example,
--   specification expressions, especially with quantifiers, belong
↪  here.
data ModelExpr where
  -- | Brings a literal into the expression language.
```

[1] https://github.com/JacquesCarette/Drasil/blob/ab9e091dabd81685ddef86b0d218582c9f75cb20/code/drasil-lang/lib/Language/Drasil/ModelExpr/Lang.hs#L82-L151

```
  Lit        :: Literal -> ModelExpr

  -- | Introduce Space values into the expression language.
  Spc        :: Space -> ModelExpr

  -- | Takes an associative arithmetic operator with a list of
↪  expressions.
  AssocA     :: AssocArithOper -> [ModelExpr] -> ModelExpr
  -- | Takes an associative boolean operator with a list of
↪  expressions.
  AssocB     :: AssocBoolOper  -> [ModelExpr] -> ModelExpr
  -- | Derivative syntax is:
  --   Type ('Part'ial or 'Total') -> principal part of change ->
↪  with respect to
  --   For example: Deriv Part y x1 would be (dy/dx1).
  Deriv      :: Integer -> DerivType -> ModelExpr -> UID ->
↪  ModelExpr
  -- | C stands for "Chunk", for referring to a chunk in an
↪  expression.
  --   Implicitly assumes that the chunk has a symbol.
  C          :: UID -> ModelExpr
  -- | A function call accepts a list of parameters and a list of
↪  named parameters.
  --   For example
  --
  --   * F(x) is (FCall F [x] []).
  --   * F(x,y) would be (FCall F [x,y]).
  --   * F(x,n=y) would be (FCall F [x] [(n,y)]).
  FCall      :: UID -> [ModelExpr] -> [(UID, ModelExpr)] ->
↪  ModelExpr
  -- | For multi-case expressions, each pair represents one case.
  Case       :: Completeness -> [(ModelExpr, ModelExpr)] ->
↪  ModelExpr
  -- | Represents a matrix of expressions.
  Matrix     :: [[ModelExpr]] -> ModelExpr

  -- | Unary operation for most functions (eg. sin, cos, log,
↪  etc.).
  UnaryOp        :: UFunc -> ModelExpr -> ModelExpr
  -- | Unary operation for @Bool -> Bool@ operations.
  UnaryOpB       :: UFuncB -> ModelExpr -> ModelExpr
  -- | Unary operation for @Vector -> Vector@ operations.
  UnaryOpVV      :: UFuncVV -> ModelExpr -> ModelExpr
  -- | Unary operation for @Vector -> Number@ operations.
  UnaryOpVN      :: UFuncVN -> ModelExpr -> ModelExpr
```

```
  -- | Binary operator for arithmetic between expressions
↪  (fractional, power, and subtraction).
  ArithBinaryOp :: ArithBinOp -> ModelExpr -> ModelExpr ->
↪  ModelExpr
  -- | Binary operator for boolean operators (implies, iff).
  BoolBinaryOp  :: BoolBinOp -> ModelExpr -> ModelExpr ->
↪  ModelExpr
  -- | Binary operator for equality between expressions.
  EqBinaryOp    :: EqBinOp -> ModelExpr -> ModelExpr -> ModelExpr
  -- | Binary operator for indexing two expressions.
  LABinaryOp    :: LABinOp -> ModelExpr -> ModelExpr -> ModelExpr
  -- | Binary operator for ordering expressions (less than, greater
↪  than, etc.).
  OrdBinaryOp   :: OrdBinOp -> ModelExpr -> ModelExpr -> ModelExpr
  -- | Space-related binary operations.
  SpaceBinaryOp :: SpaceBinOp -> ModelExpr -> ModelExpr ->
↪  ModelExpr
  -- | Statement-related binary operations.
  StatBinaryOp  :: StatBinOp -> ModelExpr -> ModelExpr ->
↪  ModelExpr
  -- | Binary operator for @Vector x Vector -> Vector@ operations
↪  (cross product).
  VVVBinaryOp   :: VVVBinOp -> ModelExpr -> ModelExpr -> ModelExpr
  -- | Binary operator for @Vector x Vector -> Number@ operations
↪  (dot product).
  VVNBinaryOp   :: VVNBinOp -> ModelExpr -> ModelExpr -> ModelExpr


  -- | Operators are generalized arithmetic operators over a
↪  'DomainDesc'
  --   of an 'Expr'.  Could be called BigOp.
  --   ex: Summation is represented via 'Add' over a discrete
↪  domain.
  Operator :: AssocArithOper -> DomainDesc t ModelExpr ModelExpr
↪  -> ModelExpr -> ModelExpr
  -- | A different kind of 'IsIn'. A 'UID' is an element of an
↪  interval.
  RealI    :: UID -> RealInterval ModelExpr ModelExpr -> ModelExpr

  -- | Universal quantification
  ForAll   :: UID -> Space -> ModelExpr -> ModelExpr
```

Source Code A.8: Current ModelExpr Constructor Encoding (TTF)[1]

```haskell
class ModelExprC r where
  -- This also wants a symbol constraint.
  -- | Gets the derivative of an 'ModelExpr' with respect to a
↪  'Symbol'.
  deriv, pderiv :: (HasUID c, HasSymbol c) => r -> c -> r

  -- | Gets the nthderivative of an 'ModelExpr' with respect to a
↪  'Symbol'.
  nthderiv, nthpderiv :: (HasUID c, HasSymbol c) => Integer -> r ->
↪  c -> r

  -- | One expression is "defined" by another.
  defines :: r -> r -> r

  -- | Space literals.
  space :: Space -> r

  -- | Check if a value belongs to a Space.
  isIn :: r -> Space -> r

  -- | Binary associative "Equivalence".
  equiv :: [r] -> r

  -- | Smart constructor for the summation, product, and integral
↪  functions over all Real numbers.
  intAll, sumAll, prodAll :: Symbol -> r -> r
```

[1]https://github.com/JacquesCarette/Drasil/blob/ab9e091dabd81685ddef86b0d218582c9f75cb20/code/drasil-lang/lib/Language/Drasil/ModelExpr/Class.hs#L30-L51