ADDING TYPES AND THEORY KINDS TO DRASIL

# ADDING TYPES AND THEORY KINDS TO DRASIL

By JASON BALACI, B.Sc.

A Thesis
Submitted to the Department of Computing and Software
and the School of Graduate Studies
of McMaster University
in Partial Fulfillment of the Requirements
for the Degree of
Master of Science

Master of Science (2023) McMaster University
(Department of Computing and Software) Hamilton, Ontario

TITLE: Adding Types and Theory Kinds to Drasil
AUTHOR: Jason Balaci, B.Sc.
SUPERVISOR: Dr. Jacques Carette
PAGES: xvi, 92

# Lay Abstract

Drasil is a framework for generating software artifacts, such as code and documentation, that has great potential for improving software quality. Drasil focuses on generating Scientific Computing Software (SCS) from a Software Requirements Specification (SRS) template where it has been shown to improve software traceability, verifiability, and reproducibility, and knowledge reusability. However, Drasil faces issues with using inputted scientific theories for code generation, handling invalid mathematical expressions, and carrying all the different types of data we want to input into it. This work focuses on 4 areas in Drasil to help it realize its full potential: (1) making theories more usable for code generation by defining their structure, (2) splitting up the expression language so that we can restrict terms to specific contexts (such as code, computation, and general discussion), (3) create a system of type rules and automatically check certain expressions against them, and (4) unlock Drasil's database to store all kinds of data.

# Abstract

Drasil is a software suite for generating software, with a particular focus on generating Scientific Computing Software (SCS) following the requirements described in an abstract Software Requirements Specification (SRS) template. The template breaks up scientific knowledge into various categories, and the abstracted variant of the template makes it digestible for Drasil. A series of DSLs are used to "fill in" the template, from which Drasil is able to interpret an instance of, and configure a generation procedure to generate software. The template's theory encodings contain a shallow depth of knowledge, limiting how many ways we can interpret them. To begin strengthening this depth, we create a structure that concretely outlines Drasil's currently encoded theory kinds, allowing us to create more domain-specific interpretation opportunities for them. Similarly, each theory kind contains a particular subset of mathematical language that is relevant to them, and we act on this information to restrict usable expression terms to their related contexts. To further enrich the admissibility of expressions, we also make one of the most critical subsets, that for concrete theory transcription, type-safe by building a bidirectional type-checker and system of type rules. The type-checker shows considerable success highlighting previously undiscovered instances of ill-typed expressions in Drasil's case studies. Finally, as Drasil relies on a plethora of different types of knowledge, it needs a place to store them. Thus, we create a system to store any instance of any type of knowledge in Drasil's memory bank of knowledge by creating a universal type carrier.

# Acknowledgements

First and foremost, I would like to thank my supervisor, Dr. Jacques Carette. Through his invested time, effort, and guidance, I was able to grow as a researcher, explore my ideas, and learn more than I could have hoped. Additionally, I would like to Dr. Spencer Smith for the encouraging discourse, feedback, and advice in our Drasil meetings. Furthermore, I would like to thank Dr. Wolfram Kahl for his advice and feedback on my work, and Dong Chen for his help with editing my thesis. Thanks to my parents Yakup and Christina, my brother Daniel, and my aunt Leila for their endless love and support of me while I continue my education. Thanks to my friends for always being there for me. In particular, I would like to thank Hala for her support and encouragement throughout this process.

Finally, I would also like to thank the Drasil research team present and past for the memories and fruitful discussion.

I could not have done this without any one of you.

# Contents

# List of Figures

# List of Tables

# List of Source Codes

# List of Abbreviations and Symbols

| | |
|---|---|
| **ADT** | Algebraic DataType |
| **AML** | Algebraic Modelling Language |
| **API** | Application Programming Interface |
| **AST** | Algebraic Syntax Tree |
| **CMS** | Content Management System |
| **CPU** | Central Processing Unit |
| **CSV** | Comma-Separated Values |
| **DSL** | Domain-Specific Language |
| **FFI** | Foreign Function Interface |
| **GADT** | Generalized Algebraic DataType |
| **GHC** | Glasgow Haskell Compiler |
| **GOOL** | Generic Object-oriented Language |
| **GUI** | Graphical User Interface |
| **HTML** | HyperText Markup Language |
| **IM** | Instance Model |
| **JSON** | JavaScript Object Notation |
| **JVM** | Java Virtual Machine |
| **KMS** | Knowledge Management System |
| **NASA** | National Aeronautics and Space Administration |
| **ODE** | Ordinary Differential Equation |
| **OO** | Object-Oriented |
| **RDBMS** | Relational Database Management Systems |
| **SCS** | Scientific Computing Software |
| **SIS** | Software Interface Specification |
| **SQL** | Structured Query Language |
| **SRS** | Software Requirements Specification |
| **TTF** | Typed Tagless Final |
| **UID** | Unique Identifier |
| **WYSIWYG** | What You See Is What You Get |
| **DblPendulum** | Double Pendulum |
| **GamePhysics** | Game Physics |
| **GlassBR** | Glass Breaking |
| **HGHC** | Heat Transfer Coefficients between Fuel and Cladding in Fuel Rods |
| **NoPCM** | Solar Water Heating System Without PCM |

| | |
|---|---|
| **PDController** | Proportional Derivative controller |
| **Projectile** | Projectile |
| **SglPendulum** | Single Pendulum |
| **SSP** | Slope Stability analysis Program |
| **SWHS** | Solar Water Heating System |

# Reading Notes

Before reading this thesis, I encourage you to read through these notes, keeping them in mind while reading.

- Drasil's source code is publicly available on GitHub, and Drasil's documentation (user-facing, and internal) is available on the Drasil project homepage. Drasil's public wiki is hosted on the same GitHub repository, containing information on potential future Drasil projects, Drasil-related papers, a developer workspace configuration and "quick start" guide, and a guide for building your own project with Drasil. Similarly, the source code for this thesis is also publicly available.
- "Source Code" snippets with "Original" in their title show code snippets as they appeared *before* this work. All other snippets are either pseudocode or a view of the actual code at a particular git blob.
- The source code related to the prototyped `ChunkDB` relevant to Chapter 8 is also publicly available.
- Please note that blue coloured text in a monospaced font (such as `ExampleText`) refers to names you can find in Drasil's source code.
- At times, we will refer to "software artifacts" as just "artifacts."
- When we refer to "Haskell," we are referring to the Haskell 2010 specification [1] and/or Haskell code compiled by GHC 8.8.4 [2].
- This report is available in two (2) flavours: one intended for viewing on a computer (the default), and one intended for printing. The one intended for viewing on a computer will have hyperlinks coloured in red and does not show website references explicitly. In particular, all "Source Code" content will link directly to a snippet of code. The copy intended for printing will show website references explicitly by adding footnotes to hyperlinks.

# Declaration of Academic Achievement

Unless otherwise stated (through citation or otherwise), this thesis is the work of my own (Jason Balaci). In particular, through Dr. Jacques Carette's project aims, direction, and supervision, I was able to complete this work.

This thesis contributes to Drasil, a research project principally investigated by Dr. Jacques Carette and Dr. Spencer Smith, and contributed to by many fellow students before me.

# Chapter 1

# Introduction

Software developers pull on their understanding of problems to build software solutions, and working together, developers similarly pull their understanding from a shared pool of knowledge. Often, developers share their knowledge through documentation, keeping every product owner[1] "in the loop." As knowledge and requirements change, software implementations trail behind the current needs and understandings, until developers manually update the code. By applying generative techniques to software development, we may have some, or *all*, of the manual updates performed for us.

Drasil [3] is an exploration of applying generative techniques to well-understood domains to create an alternative way of developing software. Focused on building Scientific Computing Software (SCS), Drasil allows users to describe scientific problems using an abstracted Software Requirements Specification (SRS) template to automatically build a software solution conforming to the specifications outlined. However, Drasil is only capable of generating software for the problems that it has sufficiently "understood" (i.e., ones that have been sufficiently dissected and encoded in Drasil).

With a focus on scientific theories and mathematical expressions, in this thesis, we aim to make some improvements to how Drasil captures certain kinds of knowledge. However, we will also make improvements to how Drasil stores knowledge in general.

## 1.1 Background: Drasil

Drasil is a software suite for generating software from well-formed, principled "stories"[2]. Focused on Scientific Computing Software (SCS), Drasil allows users to "fill in the blanks" to describe their scientific problems using a precise Software Requirements Specification (SRS) format [4]. By providing sufficient information in the "blanks," Drasil is able to use the information to generate various software artifacts, including whole programs (in various supported languages, such as Java and C#), build tools, and documentation. The "blanks" are holes for *domain-specific knowledge* and are filled in using one of many Domain-Specific Languages (DSLs). Each individual fragment of knowledge in Drasil is known as a "chunk," and we encode each useful idea necessary to discuss/build our

[1] For our purposes, defining a "product owner" as all people that have any responsibility in the development of a software artifact.

[2] For our purposes, "stories" being an abstraction of the requirements of the generated software.

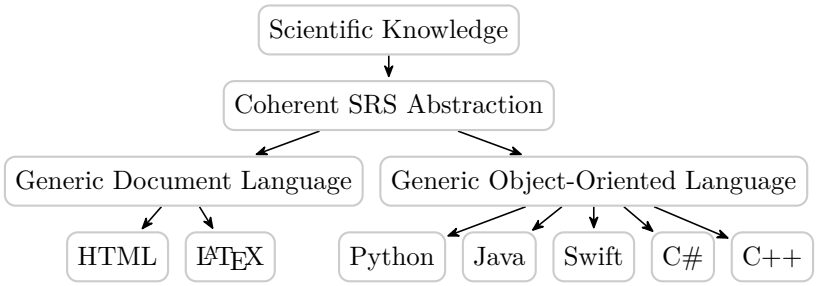
desired software artifacts as chunks. For example, if we wanted to encode a variable, $\theta_c$, representing the firing angle of a cannon[3], we might write:

Source Code 1.1: Pseudocode: Example Angle Variable Encoding

```
firingAngle = makeUCWDS "desiredFiringAngle"
    (cn "desired firing angle")
    (S "needed firing angle for a cannon to hit a target")
    (sub lTheta lC)
    radian
```

Source Code 1.1 encodes the variable with a UID ("desiredFiringAngle"), a short name ("desired firing angle"), long name ("needed firing angle for a cannon to hit a target"), a symbol ($\theta_c$), and a unit (radians). The many used chunks and DSLs make up a *network of domains* [5], which allow Drasil to make domain-specific transformations, such as the one most desired in Drasil: generating computation software conforming to a precise SRS.

Figure 1.1: Rough Sketch of Network of Domains Relevant to the *Smith et al.* SCS Generator




Roughly, Figure 1.1 shows how Drasil's *Smith et al.* knowledge transformer works, with each node representing a domain of knowledge, and each arrow representing a mapping between them. Drasil users mostly enter in their scientific knowledge near the "top" of the diagram to form a coherent SRS abstraction using relevant DSLs. Drasil takes their SRS abstraction, audits it, and allows the user to pick from a series of options to generate software that conforming to the scientific problem encoded.

The "scientific knowledge" at the top of the diagram is not necessarily a complete capture of all scientific knowledge, of course. Rather, it is a "bubble"/collection of the scientific knowledge. Drasil's development follows the needs of a series of manually built case studies, using them as "seedling" data to navigate development. As Drasil is "taught" more information[4], Drasil's span of producible artifacts widens. Unfortunately, at the moment, Drasil is not able to generate code for all of its case studies.

Some case studies are currently in-progress (i.e., incomplete) and won't be a focus of this work, but left for future work by others. Instead, we will focus on those which *we*

[3]Running example based on Drasil's Projectile case study.

[4]By providing it with a means of discussing relevant ideas through DSLs.

understand how to manually produce code for. Specifically, we will largely focus on a common denominator: strengthening mathematical knowledge capture, but we will also spend time learning how to scale Drasil's knowledge (chunk) database against expansion in further chunk type creation. Additionally, we will primarily pull examples from the Projectile, GlassBR, and DblPendulum case studies.

### 1.1.1 Theories and Expressions

One of the most important aspects of understanding and describing scientific problems is the underlying *theories*. Drasil relies on users describing theory knowledge using *relations* through a *single universal untyped mathematical expression language*. To be clear, this means that Drasil relies on on-demand interpretation and comprehension of *terms* from this expression language. Equations, relations, derivations, constraints, and definitions are all described using this single language. However, the language does not contain sufficient "depth" to adequately make use of its encoded information. In other words, this expression language is a "lower-level" view of the information we really need to make domain-specific transformations.

Drasil attempts to use the *theories* gathered from a user-filled SRS to understand what problem they intended to describe (i.e., calculating the inputs and outputs, and a sequence of calculation steps that go from the former to the latter). Each theory needs to be sufficiently understood to produce calculation steps that we can use. For example, if we wanted to define our firing angle variable (Source Code 1.1) in terms of the distance of the target, gravity, and the initial velocity[5], we might write: $\arcsin\left(\mathit{targetDistanceFromCannon} \cdot g/v^2\right)/2$. In Drasil, we might encode this as Source Code 1.2, where an `InstanceModel`[6] carries information about the theory (e.g., a derivation, a defined quantity, and a defining relation of the quantity).

Source Code 1.2: Pseudocode: Example Angle Equation Encoding

```
firingAngleIM :: InstanceModel
firingAngleIM = imNoRefs firingAngleEqn
    [] (qw firingAngle) [UpFrom (Exc, dbl 0)]
    (Just firingAngleEqnDerivation)
    "firingAngleFormula" []

firingAngleEqn :: RelationConcept
firingAngleEqn = makeRC "firingAngleEqn"
    (nounPhraseSP "firing angle formula")
    (sy firingAngle $= arcsin (sy targetDistanceFromCannon * sy gravity /
    ↪  square (sy v)) / dbl 2)
```

[5] Assuming no air drag or resistance.

[6] An instance of a concrete theory usable in the solution to a scientific problem, modelled after "instanced models" [4].

With this theory encoding, we have enough information to generate a visually pleasing description box for the SRS LaTeX and HTML documents (Source Code A.1). Converting to typesetting languages is reasonably unproblematic because they only use the theory knowledge in a shallow light (displaying) and we can create any glyph we might need. More importantly (and relevant to this thesis), we can also convert this theory to Python code as well, using the encoded defining relation of the firing angle in Source Code 1.2:

Source Code 1.3: Pseudocode: Example Angle Equation Python Conversion

```
## \brief Calculates desired firing angle: needed firing angle from a
 ↪  cannon to hit a target (radians)
# \param inParams structure holding the input values
# \param g gravitational acceleration (m/s^2)
# \return desired firing angle: needed firing angle from a cannon to hit
 ↪  a target (radians)
def func_theta_c(inParams, g):
    return math.asin(inParams.targetDistanceFromCannon * g / (inParams.v
     ↪  ** 2)) / 2
```

However, this example is reasonably simple. If we were handed an equation of this $\nu = f(x, y, z)$ form, it's somewhat reasonable, but perhaps dubious to assume that one way to define $\nu$ is $f(x, y, z)$. However, if we were handed an equivalent, implicit version of it, then we can assume less information about it. For example, if we re-wrote the equation for Source Code 1.2 as $\mathit{targetDistanceFromCannon} \cdot g = v^2 \sin(2\theta_c)$, then we've lost the critical *definition* assumption that we used to write the python code. Automatically isolating for $\theta_c$ is possible, but we're interested in the definitional information, and we want it as readily-available as possible. Now, with this scrambled variant as well, we similarly can't typeset it in the expected definitional form we desired, even though the relation is equivalent. With more complex theories, more issues arise. For example, defining theories with multiple definitions lacks information about which definition to "pick" for software implementations, ODEs lack information about solving methods, and some theories may be purely abstract and entirely unusable for code generation. In other words, the current encoding of theories lacks information about the structure of the theories. Additionally, we also lack information about when expression terms are usable in the context of code (e.g., expressions involving derivatives are not always directly usable in code), where we expect them to be directly translatable to programming languages. As a result, knowing *how to* and *when you can* transform captured theories into other forms (such as executable code) is a complex task (similar to the complexity associated with transpiling a general-purpose program into another — an exercise in futility!).

With the theory encoding above, we also never discussed what *targetDistanceFromCannon*, $g$, nor $v$ were. On paper and pencil, it's fairly reasonable to assume that they are all some numeric type. However, when we generate software, we expect it to be *executed*, and hence undergo a type-checking phase by compilers/interpreters. As the expression language is currently untyped, Drasil currently allows software generation with "ill-typed" (according

to the respective compilers and interpreters) expressions. In particular, for example, this poses problem for Swift code generation because Swift doesn't allow numeric addition if both operants are not of the same type, while Java uses implicit type coercion to mask these issues. Ultimately, this means that more time is spent developing and debugging generated artifacts than we would like. We would prefer that generated software come with an assurance of correctness.

### 1.1.2 Capturing (& Remembering) Everything

Knowledge capture is at the heart of Drasil. It is what allows Drasil to make domain-specific interpretations that we're interested in, to ultimately generate software artifacts. *Chunks* are unique instances of knowledge, tagged with a UID for referencing them, and a *type* for understanding their structure. The chunk is unique with respect to a global "chunk database" that Drasil registers all chunks into. At the moment, Drasil's chunk database uses a series of typed maps (Source Code 8.1) with UID keys and typed chunk values. This means that as we create new types of chunks in Drasil, the list of maps in the database will grow. Additionally, the list of types will have to be known before using Drasil, which means that extending the list of chunks is difficult for users. In other words, the current chunk database is not *extensible* because of its typed nature.

## 1.2 Problem Statement

Drasil isn't able to adequately make domain-specific interpretations of encoded theories because it lacks structural information about them. They also don't provide static information about the contexts in which they are usable, partially because the underlying expression language exposes little information about this too. Additionally, Drasil's lack of type information stops us from reliably generating usable software (in particular, there is no assurance of type checking). Together, these issues affect our ability to reliably and flexibly generate software. Finally, Drasil's chunk database is not extensible, limiting admissible chunk types to only those that exist in core Drasil.

**Research Questions**:

**RQ1** Drasil's current encoding of "theories" are essentially black boxes. We would like to use structural information present in the short list of the "kinds" of theories that show up in scientific computing. How do we codify that?

**RQ2** Drasil's theory encodings rely on a single mathematical expression language, which does not expose information about applicability to different contexts. In each context (e.g., code, theories, and common arithmetic), certain terms of the expression language should be treated differently or are simply inapplicable. How can we restrict term usage by context?

**RQ3** How can we ensure that our mathematical expression language admits only valid expressions?

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

## 1.3 Contributions of the Author

In listed code snippets with "Original" in their titles, I'm referring to code that existed in full before I started contributing to Drasil. As such, I claim no authorship in those snippets. Unless otherwise stated, all other snippets, excluding excerpts from generated artifact snippets, includes some of my work, but might also include the work of others who were also contributing to the project while I was also actively contributing. The work of others might include, but is not limited to, code formatting, code commenting, and extensions.

My notable contributions include:

- Implementing `ModelKinds` to expose theory structure knowledge, creating opportunity for new domain-specific interpretation of theory knowledge (resolving RQ1). The solution builds on a prototype by Dr. Jacques Carette[7].
- Splitting the mathematical expression language into 3, each with their own intended area of usage, but using a TTF [6] to keep same user-friendliness in expression transcription (resolving RQ2).
- Adding type checking and inference to mathematical expression languages (resolving RQ3).
- Creating an extensible chunk database prototype that can register chunks with types external to core Drasil and chunks with type parameters (resolving RQ4).

## 1.4 Thesis Outline

In Chapter 2, we discuss what we can learn from the source code of general software, and specifically, how we can abstract over them to form similar software, varying over certain aspects of the software to make the background knowledge reusable. Chapter 3 discusses Drasil, a software suite for generating software from coherent descriptions of scientific problems. Chapter 4 discusses how Drasil captures and uses theories to generate code, and how we improved theory capture to create more opportunity for domain-specific interpretation of the theories (RQ1). Chapter 5 discusses how Drasil encodes mathematical expressions and associated issues (RQ2). Chapter 6 revisits the existing captured theories in Drasil, rebuilding them with specialized encodings for others to use as needed. Chapter 7 discusses issues associated with the formation of mathematical expressions and what it means for expressions to be "well-typed" (RQ3). Chapter 8 focuses on how Drasil stores information, and how it can scale against future development of Drasil and libraries (RQ4). Chapter 9 discusses future work. Finally, Chapter 10 sums up the achievements of this work.

[7] Unfortunately, the code associated with the prototype had been deleted. However, Source Code 4.8 is, if memory serves me correctly, nearly identical.

# Chapter 2

# Ideology[1]

"Generation" is at the heart of this work. However, unlike GitHub and OpenAI's Copilot [7], we don't delve into artificial intelligence. Copilot uses AI to autocomplete code using smaller snippets and comments, while we focus on capturing the meaning of specific subsets of language to generate software artifacts through description of their requirements. In other words, we focus on encoding the important bits of information that we use to discuss requirements of software and how they relate to the software artifacts we would normally manually write. We focus on capturing knowledge through codifying well-understood fragments with DSLs to capture the *harmonious relationship* between *knowledge* and *software artifacts* to improve software *maintainability* and knowledge *communication* and *reusability*.

## 2.1 On Developing Software

As software developers, we encode "stories" through software. As an example, programs made to process CSV files tell a story of how data can be entered, adjusted, and output. Compilers tell a story of how human-readable programs can translate to various assembly languages. In these stories, we commonly use similar terminology and ideas. Thankfully, we avoid writing "a lot of the same code" by abstracting over variables and sharing reusable code through libraries. With libraries, we're able to share our knowledge with our future selves and others alike. Once we've reasonably stabilized our libraries, it becomes a large gain in the reusability of our efforts —we don't need to worry about making the same bugs twice! However, a few issues arise: the code might become out-of-date (or out-of-sync), others might not understand what we wrote (and thus not trust/use it), or we might need to use the same conceptual ideas but in a different programming language. If our understanding of key ideas changes, we might need to perform large, manual refactoring of our code, and also update our documentation too. While we might write "idiomatic" code, reverse engineering code is tedious, and even then, we can only analyze the code we *see*, and not what knowledge it took to write that code. Finally, we might be able to write a Foreign Function Interface (FFI), but they're often brittle and demanding of our time, due to initial and repeated complex analysis for each library

[1] Or, at least, my understanding of a software development ideology I believe to be relevant to this work and Drasil.

update. We might even be forced to use particular programming languages.

Often, we look to using mature libraries and frameworks to underpin our projects, but usually without a guarantee that how we use and connect libraries is "safe." For example, the sinking of the Vasa ship [8] was 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 unbalanced weight distribution, contributing to its demise. Similarly, when the Mars Climate Orbiter travelled to Mars, it met an early demise due to a navigation issue [9]. Lockheed Martin built the orbiter ground controller software, but it didn't conform to NASAs Software Interface Specification (SIS). The commands sent from Earth used English units (specifically, pound-seconds) while the orbiter assumed that it would receive commands using the metric system (Newton-seconds). As such, the orbiter missed its intended target orbit altitude, falling into the Martian atmosphere, and ultimately disintegrating due to atmospheric stress.

Experts had an in-depth understanding of the "story" of each project, with sound rationale for how things worked and should have worked, and yet both ended in misfortune. Of course, most software is not critical, and issues in most software will not result in an orbiter disintegrating in Martian atmosphere or a ship sinking, but there is something that we can learn: *communication* and *synchronization* of development efforts is vital for building reliable solutions.

While we as developers don't often build and connect physical things, we do connect pieces of code[2], and misunderstandings of tacit project knowledge occur too. Thus, we believe we need to revisit our original sensation when we recognized code duplication and decided to make reusable components.

The reason is obvious[3]! We felt this sensation because we already had a mental model that connected some key concepts to some code we wrote, so we decided to make reusable views (code) of our knowledge. However, the code is only a shallow view of our knowledge, containing very little discussion of the conceptual underpinnings and the role they play in the greater "story." Shallow views of implicit, unwritten knowledge, unfortunately, does not come with guarantee of harmony with the way it's used.

## 2.2 Dreams of Generation

Unlike the Vasa, for stories where the desired end-product somehow involves software, we can remedy the communication issue partially by unifying it under one cohesive story. Software Requirements Specifications (SRSs) play a large role in unifying communication of software needs. However, the communication and maintained synchronization of the software requirements into the final software product is still brittle (as evidenced by the Mars orbiter), as it remains heavily reliant on manual labour to translate it into software artifacts. In other words, the translation from our knowledge (the important part) is laborious and prone to error, and hence, not simple enough. So, now we wonder: how can we simplify the process?

[2]Though robotics is a field too.

[3]Ignoring the more obvious reasons, such as copy-pasting code, or mere awareness of textual similarities.

Our end-goal should be "assembly-line" style engineering of software [10], free of logical issues. To attain this, we need to have clear criteria for what it means for our software artifacts to be free of logical issues. However, to do this, we need to discuss relevant models. For example, if we're interested in the accuracy of a bank accounts cached balance, we discuss *the user*, the relevant *transactions*, and then a *validation algorithm*. Realizing this example in code might have us retrieve a user's bank transactions, calculate the expected balance, and compare it with a cached balance. The code only contains one dimension of this discussion: the actions. It has no understanding of the substance, nor how it can be similarly used in other scenarios. To audit the code, we analyze the code ourselves, potentially with extra testing tools to make things quicker, but in general, it relies on us and our understanding. To remedy this, we look to describing software as we've done here: as a "view" of some story or discussion. In other words, we want to build software artifacts through description as opposed to manual conversion of description to artifact.

### 2.2.1 The Goal

If we think of a Java compiler as a sort of *generator* of JVM bytecode, we can think of the Java programs as the instructions and inputs to the generator. We rarely look at the actual bytecode ourselves, but we do have confidence in knowing what it will do when executed. Now, we look to go one "level" up. In other words, we look to inputting our understanding of particular applications through another generator (an abstraction level up) to somehow obtain code for it.

Especially in the cases where everything is "well-understood" [10][4], we want to focus on communicating problems and how solutions solve them so that we can *generate* usable solutions (a software artifact), and keep them up-to-date through re-generation.

### 2.2.2 Reconciliation

By focusing on capturing well-understood [10] knowledge, we can use (and re-use) knowledge across specialized generators to generate software for specific kinds of problems. For example, statisticians frequently use and discuss various kinds of distributions, such as "Poisson," "Uniform," and "Normal," and when they do, they're typically familiar with their parameters, expectations functions, and how to use them to estimate likelihoods. Hence, by focusing on using *DSLs*, we can build specialized interpreters for them. Furthermore, by connecting them in precise manners (with similar precise languages), we can form large meaningful *networks of domains* [5] that form our well-understood problem spaces. For well-understood problem spaces, we can compose a series of domain-specific interpreters. With enough effort, we could take a whole "problem description" that draws in multiple fields, and generate software that somehow "solves" it.

By switching our focus of manual software development to manual problem description and relationships to "solutions," we shift where we can make bugs, and how they propagate. Namely, logical bugs will occur more than once so as long as the same knowledge pulled from is drawn more than once. Thus, each logical bug should be more visible

[4] Irrelevant of rarity!

and easier to spot. Additionally, the generated software becomes directly *traceable* to its logical foundations. Hopefully, with adequate dissection of related concepts, bugs should also be less intricate. Furthermore, generated artifacts are simpler to *maintain* (i.e., kept in synchronization) with its related knowledge-base and story by *regenerating* it. Finally, as opposed to sharing *code*, in this paradigm, we *communicate knowledge*, achieving language-agnostic reusability, focusing on sharing meanings and families of problems rather than solutions. Thus, by focusing on generating from meaningful descriptions, we obtain *knowledge reusability* (as opposed to *code* reusability), and increased software *maintainability*, *reliability*, and *traceability*. Of course, generating all software artifacts appears grandiose, and, perhaps, reductive or ignorant of many difficulties in software development. Thus, we must discuss feasibility[5].

### 2.2.3 Feasibility

For us to discuss feasibility of this idealized development paradigm, we must discuss the *depth* and *breadth* of knowledge we need to make this feasible[6]. Depth of knowledge refers to the vertical knowledge understood about a specific fragment of knowledge, and its preciseness. For example, we may have a low-depth of knowledge and claim that English sentences are a sequence of characters. Alternatively, 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.

#### Low Depth & Narrow Breadth

At a shallow depth and narrow breadth of knowledge capture, this paradigm is very practical, and already heavily used. Widely used Content Management Systems (CMSs), such as WordPress [11] and Drupal [12], and web frameworks, such as Django [13] and Laravel [14], are arguably also following similar ideals as this ideology. Notably, they *deeply embed* [6] knowledge in their frameworks and libraries using their host programming languages basic features. They all typically provide a basic understanding of "user's" of a hosted website, facilities to write HTML content in one way or another (e.g., WYSIWYG editors, templates, and plugins). While some might be, these listed above are not specific to one specific use-case. They're versatile products, usable for a wide variety of use-cases because they ship with low but sufficient depth of knowledge[7] such that you can use them for a wide variety of different applications (e.g., blogging, ticketing, booking, accounting, etc.). 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

[5] You may skip the remainder of this chapter if you so wish, it is not strictly required to understand the rest of my work, but it doesn't hurt.

[6] Note that "knowledge" is captured through codifying DSLs.

[7] They might call it "features."

contained within them. Through this, end-users may 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.

The mechanized generation-related components of the ideology is also fairly shallow in this area, but still, highly feasible. In some sense, almost any individual instance of "generation" is an area of low depth and breadth as well.

At the lowest depth and narrowest breadth of knowledge capture, we aren't really capturing any meaning. Rather, we are programming our software artifacts directly. Hence, this area is already very feasible, evidenced by the usefulness of software in usage today, and the way said software was made.

#### Specialized Tools: Deep Depth & Narrow Breadth

We might think of software that captures deep knowledge about a specific topic as *specialized tools*. Tailored to specific needs, they come with extra features for highly specific, potentially niche, categories of problems[8]. These already exist, and are similar to the "Bottom," highly practical. For example, Algebraic Modelling Languages (AMLs) (such as HashedExpression [15]) allow you to describe complex mathematical algorithms and generate optimized software tools for solving them. Another example is parser generators (such as Happy [16]). They capture complex information about grammars and allow users to generate specialized parsers for them. When needed, specialty tools provide greater functionality than general-purpose programming languages for solving specific problems.

#### Off-the-shelf Solutions: Low Depth & Wide Breadth

Orthogonal to the "specialized tools," a capture of low depth and wide breadth of knowledge is seemingly a jack of all trades, master of none. For example, most modern programming languages come with a standard library that provide many off-the-shelf solutions to well-understood problems, but sticking to very common generic problems. Most users typically pull in a library that provides extra specialization for certain problems because the standard library didn't go deep enough for their needs. Similar to the categories before, these are widely used and similarly practical.

#### Utopia: Deep Depth & Wide Breadth

Finally, we've reached the category of "deep depth and wide breadth" of knowledge capture. Here, we capture the meanings of key ideas and concepts we need to build software, making as many conscious decisions explicit and visible as possible. Imagine using the specialized tools to develop every aspect of some software artifact, but for every facet of the artifact, from start to finish. This is an idealized method of developing software, where knowledge is strongly reusable and composable. Of course, this relies on heavy research on tooling. As long as developers have infinite patience and can invest infinite time

[8]Think of this category as sharp tools, such as kitchen knives. We can generally get away with using basic kitchen knives, but specialty knives help us out for particular use-cases, such as serrated or filleting knives.

into transcribing and researching to fill in gaps, anything is possible, and this ideology is very practical! Unfortunately, that situation is not quite realistic. As such, we should restrict our scope of captured knowledge to "well-understood" domains [10]. For areas of "well-understood" knowledge, this should be more feasible, merely because the discussion of key ideas is already coherent and sufficiently codified.

Here, you might use a series of DSLs to create a coherent discussion (a "story"). With it, you would have specialized interpreters built, which process it, and draw out meaningful byproducts you're interested in (such as usable software). This story can be relatively far removed from produced artifacts generating (containing potentially little to no discussion of the desired artifacts at all), or a precise description (where you might be manually writing out the final artifacts yourself). Through creating a story as a composition of many other smaller fragments of knowledge and stories, and defining interpreters as compositions of other, smaller interpreters, this is feasible. Hopefully, with enough effort, this should alleviate considerable stress associated with manual software development, by moving the focus to the important bits: the story the artifacts tell. The "quality" of the generated artifacts become a traceable reflection of the "quality" (depth and breadth) of the captured knowledge.

## 2.3 A Prospective Workflow

In theory, all "development" should occur in a Knowledge Management System (KMS), where knowledge and transformations between fragments of knowledge are transcribed. "Users" would use a system of pre-filled knowledge to piece together a template story that fits their narrative, or they would adapt an existing one to fit theirs.

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). They are the ones that have an issue that can be resolved with some sort of software artifact. At the "top," product owners designate a basic set of requirements of the artifacts using a coherent *formal description*. Product designers/orchestrators will take the requirements and convert them into a coherent *story* for how the requirements may be translated into a final product. The story builds on well-understood knowledge of various domains encoded by domain experts. 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.

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

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.

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

produced software artifact.

### 2.3.1 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 2.2.2, it is likely that the knowledge encoder will focus on writing a series of highly specific DSLs, where the languages may be restricted to as specific as one term or a handful.

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*. For the 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 [1], Agda [17], etc.) — preferring ones with a type system based on formal type theories for their feature richness.

### 2.3.2 Knowledge / Domain User & Orchestrator

The knowledge user/orchestrator is tasked with connecting the work of the domain experts into "plug-n-play" stories (arguably, compilers for the end-users to use). They should also have a working understanding of what the end-user needs, and how the needs relate to domains of knowledge. As such, they should be able to encode and reduce friction between knowledge encoded by domain experts and the goals of product owners.

### 2.3.3 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.).

Any of the three (3) user types may use the "plug-n-play" stories to describe problems and generate solutions based on them.

## 2.4 Drasil

To my understanding, Drasil [3] explores this ideology, focusing on generating scientific software from user-described scientific problems using Drasil-understood terminology (i.e., ones that a scientific domain expert previously encoded).

# Chapter 3

# Drasil

In this chapter, we will discuss the project this work contributes to, Drasil. Specifically, we will discuss at a high-level what Drasil is capable of, and how it works.

Figure 3.1: Drasil's Logo

## 3.1 What is it? What can it do?

Principally investigated by Dr. Jacques Carette and Dr. Spencer Smith, Drasil is a software suite for generating software for well-understood problems through a knowledge-first approach [3]. Drasil captures the background knowledge involved with software development to make it *reusable*, improve *maintainability* of software, and strengthen *traceability*

between desired "software artifacts"[1] and the background knowledge [18]. Currently, Drasil focuses on generating software artifacts for Scientific Computing Software (SCS), where it has been shown to improve software qualities, such as *verifiability*, *reliability*, and *usability* [19].

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. |
| 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). |

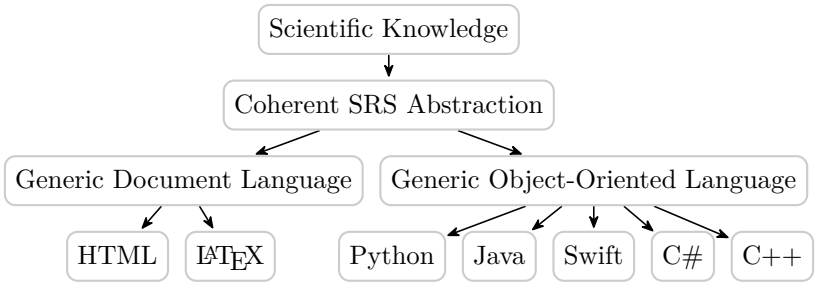
Drasil's knowledge-capture approach to software development allows users to remove themselves from the discussion of "code" and focus on the important bits: the problem the code solves and how "code" ultimately relates to it[2]. Drasil's development is navigated through a series of case studies: Table 3.1. Each case study has users input scientific problems using DSLs that build up an abstract Software Requirements Specification (SRS)

[1] "Software artifacts" being any file with a well-defined structure, such as plaintext files, Python code, LaTeX code, HTML, or JSON.

[2] Drasil allows users to "keep at a safe distance" from software, but only so far as Drasil has encoded the terminology the users rely on for conveying their problem to Drasil.

[4]. The SRS decomposes scientific problems into small structured fragments of knowledge that have direct relations to "code" and can be used flexibly as part of code generation. Through abstracting sufficient depth and breadth of the knowledge in an SRS (e.g., including the inputs, outputs, and calculation scheme), Drasil is able to generate software artifacts that satisfy the designated SRS[3] (Table 3.2). Drasil relies on the capture of a *network of domains* [5] to generate artifacts. Roughly, the network of domains related to the code generated related to an abstracted SRS is as per Figure 3.2[4], where nodes are the major categories of domains[5] and arrows are mappings between them. The current network is built according to the needs of Drasil's case studies. One notable success of the knowledge capture is the reusability of it to regenerate artifacts in different, but similarly applicable, languages. For example, the GlassBR case study had software artifacts manually built. Once the knowledge was codified in Drasil, the same knowledge allows re-creation in other languages.

Figure 3.2: Rough Sketch of Network of Domains in Drasil




Drasil is able to generate a host of Object-Oriented (OO) programming language source codes through compiling to Generic Object-Oriented Language (GOOL) [20, 21], which compiles to several OO languages (such as Java, Python, C/C++, C#, and Swift[6]). Drasil also contains renderers for printing HTML files, Makefiles, basic Markdown (enough for "READMEs"), GraphViz DOT [22] diagrams, and plaintext, LaTeX documents. SRS abstractions are renderable in either LaTeX or HTML.

Table 3.2: Drasil Case Studies Artifacts Generated

| Case Study | SRS | C/C++ | Java | C# | Python | Swift |
|---|---|---|---|---|---|---|
| GlassBR | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Projectile | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| SglPendulum | ✓ | | | | | |
| DblPendulum | ✓ | | | | | |
| GamePhysics | ✓ | | | | | |
| HGHC | ✓ | | | | | |
| PDController | ✓ | | | | ✓ | |

[3]It is helpful to think of the SRS as a sort of "recipe" that Drasil follows to navigate the generation.

[4]Note that this is not representative of all knowledge captured in Drasil, but only that which is relevant to generating code for a given SRS abstraction.

[5]In other words, each node contains its own subdomain as well

[6]Note that Swift was not discussed in [21], but the renderer was built by Brooks as well.

| | | | | | | |
|---|---|---|---|---|---|---|
| SWHS | ✓ | | | | | |
| NoPCM | ✓ | ✓ | ✓ | ✓ | ✓ | |
| SSP | ✓ | | | | | |

## 3.2 How does it work? How is it used?

As mentioned in Section 3.1, Drasil relies on building a tree of knowledge that contains sufficient information such that software artifacts can be "grown" from them. The individual pieces of knowledge are known as *chunks* and are encoded as either Algebraic DataTypes (ADTs) or Generalized Algebraic DataTypes (GADTs). Drasil, and all knowledge captured in Drasil, is deeply embedded in Haskell [1] source code[7]. Each chunk has a *type* which defines its structural information. Chunks contain information encoded with various Domain-Specific Languages (DSLs). The network of domains (roughly, Figure 3.2) is made up of a series of chunks connecting and discussing one another, similar to how we might discuss abstract concepts.

The "coherent SRS abstraction" of Figure 3.2 is modelled after the *Smith et al.* formal SRS template [4], while the "scientific knowledge" higher up is a set of interconnected chunks (and, hence, DSLs). The "scientific knowledge" chunks are used to fill in the "gaps" of the SRS template. For example, if we wanted to encode a variable, $\hat{q}_{\text{tol}}$, representing a real number, "Tolerable load," we might write it as Source Code 3.1, where it is of type `QuantityDict` (Source Code 3.2[8]), the type of variable encodings.

Source Code 3.1: Original Example Encoded Quantity: Tolerable Load

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

Source Code 3.2: Original Definition of "Quantities" (QuantityDicts)

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

[7]The source code compiles against GHC 8.8.4 [2] and uses GHC language extensions.

[8]Note that `IdeaDict`, `Space`, `Symbol` (for all `Stages`), and `UnitDefn` are coupled together to create the quantity. The components respectively define the term, type, symbol (dependent on the usage context [equational or software implementation]), and unit of the quantity.

Notably, in Source Code 3.1, the symbol, $\hat{q}_{\text{tol}}$, is built using a `Symbol` DSL. The capture of domain-specific knowledge is what sets DSLs apart from general-purpose programming languages. Domain-specific abstractions create opportunities for domain-specific *interpretation and transformation* (e.g., optimization, analysis, error checking, tool support, etc.) [5]. For example, with the symbol for "tolerable load," we have information about the structure of the symbol itself: that "q" has a "hat" and a subscript "tol." From this, we can output the same information in alternative flavours if we desired, such as plain text, or with Java-compatible naming convention (e.g., "qHatTol").

Drasil's SRS template contains more "holes"[9] for other information necessary to creating a whole "story" about how *output variables* can be calculated according to a set of *input variables* and algorithm derived through a series of *theories*. With sufficient knowledge *depth*[10] for each relevant fragment, Drasil is able to automatically "check" it for consistency and coherence, and generate representational code[11].

Unfortunately, not all of Drasil's case studies are capable of generating representational code (Table 3.2). Some case studies (GamePhysics, HGHC, and SSP) are still actively being developed, but are left incomplete at the time of writing. 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 [23] outside the normal means of Drasil's case study development. Code generation for PDController is not impossible (it's done for Python), it just requires more investigation by a domain expert for the needs of compiling to more languages. However, both the issues related to NoPCM and PDController are outside the scope of this work. In this work, we will focus on a critical common denominator between all examples: capturing mathematical knowledge for reliable SRS artifact generation. In particular, we will focus on 2 primary aspects of mathematical knowledge: the *theories* and the *expressions*.

[9] Or "blanks" if you think of the template as a "fill-in-the-blanks" puzzle.

[10] Note that the SRS template provides the *breadth* needed by design!

[11] "Representational code" meaning software that solves the problem the related SRS abstraction describes, using the algorithm outlined.

# Chapter 4

# Theories

In this chapter, we will focus on improving inspection and interpretation capabilities of theories in Drasil. Specifically, with focus on interpreting them for generating software artifacts.

## 4.1 Transforming Theories to Code

As already mentioned in Chapter 3, the SRS template [4] breaks up software requirements and problems into a series of well-understood components, providing developers with concrete solution requirements they must satisfy, and domain experts with justification for problem solutions. Notably, the SRS relates a programs *inputs* to a set of *outputs* using a set of *theories*. The inputs and outputs are sets of variables, with data that need to be somehow fed into the program, or calculated and output by the program. The theories connect the input variables to the output by forming a *solution/calculation path*. Notably, the Instance Model (IM) [4] theories together largely form the calculation path, while other background theories provide justification for them.

### 4.1.1 An Example

For example, Drasil's Projectile case study describes how to estimate if a launcher, aligned at a particular angle, will hit a target from a specific distance. The SRS requires users to fill in the:

1. input variables:
   (a) $p_{target}$, the targets distance from the launcher,
   (b) $v_{launch}$, the projectile launch speed,
   (c) and $\theta$, the launch angle.
2. output variables:
   (a) $s$, a message, explaining if the projectile hit the target, fell short, or went long,
   (b) and $d_{offset}$, the expected distance between the target position and the landing position.

3. and theories, connecting the inputs to the outputs:

   (a) $t_{\text{flight}} = \frac{2v_{\text{launch}} \sin(\theta)}{\mathbf{g}}$, estimating flight time with $v_{launch}$ and $\theta$,

   (b) $p_{\text{land}} = \frac{2{v_{\text{launch}}}^2 \sin(\theta) \cos(\theta)}{\mathbf{g}}$, a calculation of the landing position,

   (c) $d_{\text{offset}} = p_{\text{land}} - p_{\text{target}}$, calculation of distance between the targets position and the expected landing position of the projectile,

   (d) and $s = \begin{cases} \text{“The target was hit.”}, & |\frac{d_{\text{offset}}}{p_{\text{target}}}| < \varepsilon \\ \text{“The projectile fell short.”}, & d_{\text{offset}} < 0 \\ \text{“The projectile went long.”}, & d_{\text{offset}} > 0 \end{cases}$, calculating the output message.

From these 3 key bodies of information along with some supporting background knowledge (such as assumptions, constants, etc.), Drasil forms a calculation path, deriving the output variables from the input variables using the concrete theories[1]. Together, the theories form a calculation algorithm which Drasil generates representational code of. For example, Projectiles generates Source Code 4.1 for one of the Java-flavoured artifacts. In Source Code 4.1, it uses a `write_output` method to output the calculated output variables after calculating them using the relevant theories[2].

Source Code 4.1: Projectile: Java Main Method

```
/** \brief Controls the flow of the program
    \param args List of command-line arguments
*/
public static void main(String[] args) throws FileNotFoundException,
↪   IOException {
    String filename = args[0];
    double g_vect = 9.8;
    double epsilon = 2.0e-2;
    double v_launch;
    double theta;
    double p_target;
    Object[] outputs = get_input(filename);
    v_launch = (double)(outputs[0]);
    theta = (double)(outputs[1]);
    p_target = (double)(outputs[2]);
    input_constraints(v_launch, theta, p_target);
    double t_flight = func_t_flight(v_launch, theta, g_vect);
    double p_land = func_p_land(v_launch, theta, g_vect);
```

[1]The “concrete theories” are the “instanced model” and “data definitions” [4] encodings (respectively, `InstanceModel`s and `DataDefinition`s) which are somehow usable in the solution of a scientific problem.

[2]Note: $t_{\text{flight}}$ is seemingly unused in the generated code, but it is used in the derivation of $p_{\text{land}}$. However, it being “unused” is irrelevant to this work.

```
    double d_offset = func_d_offset(p_target, p_land);
    String s = func_s(p_target, epsilon, d_offset);
    write_output(s, d_offset);
}
```

As seen in Source Code 4.1, it relies on a few methods calculating the related "instance models" on its behalf. Those methods are a specific *interpretation*[3] of the theory knowledge. Specifically, it is a *calculation-focused* interpretation. For example, to calculate $p_{\text{land}}$, Source Code 4.1 relies on the method `func_p_land` to calculate the landing position. Source Code 4.2 is one possible exported method for calculating $p_{\text{land}}$ given $v_{\text{launch}}$, $\theta$, and $\mathbf{g}$.

Source Code 4.2: Projectile: Java $p_{\text{land}}$ Calculation Method

```
/** \brief Calculates landing position: the distance from the launcher to
 ↪  the final position of the projectile (m)
    \param v_launch launch speed: the initial speed of the projectile
     ↪  when launched (m/s)
    \param theta launch angle: the angle between the launcher and a
     ↪  straight line from the launcher to the target (rad)
    \param g_vect gravitational acceleration (m/s^2)
    \return landing position: the distance from the launcher to the final
     ↪  position of the projectile (m)
*/
public static double func_p_land(double v_launch, double theta, double
 ↪  g_vect) {
    return 2 * Math.pow(v_launch, 2) * Math.sin(theta) * Math.cos(theta)
     ↪  / g_vect;
}
```

Of course, to generate this method in Source Code 4.2, Drasil relies on a sufficient capture of its underlying knowledge, and a means of transforming said knowledge to "code" (e.g., sufficient information that answers: What do we want to define? How can it be converted to code?). This capture of theories is done using Drasil's representation of an "Instance Model." To build it, Drasil uses information gathered from what users fed in about it, via Source Code 4.3.

Source Code 4.3: Original $p_{\text{land}}$ Theory Definition

```
landPosIM :: InstanceModel
landPosIM = imNoRefs landPosRC
```

[3] Or "view."

```
  [qwC launSpeed $ UpFrom (Exc, 0),
   qwC launAngle $ Bounded (Exc, 0) (Exc, sy pi_ / 2)]
  (qw landPos) [UpFrom (Exc, 0)]
  (Just landPosDeriv) "calOfLandingDist" [angleConstraintNote,
  ↪  gravitationalAccelConstNote, landPosConsNote]

landPosExpr :: Expr
landPosExpr = sy landPos $= 2 * square (sy launSpeed) * sin (sy
↪  launAngle) *
                                cos (sy launAngle) / sy
                                ↪  gravitationalAccelConst

landPosRC :: RelationConcept
landPosRC = makeRC "landPosRC" (nounPhraseSP
↪  "calculation of landing position")
  landPosConsNote landPosExpr

landPosDeriv :: Derivation
landPosDeriv = mkDerivName (phrase landPos) (weave [landPosDerivSents,
↪  map E landPosDerivEqns])

landPosDerivSents :: [Sentence]
landPosDerivSents = [landPosDerivSent1, landPosDerivSent2,
                     landPosDerivSent3, landPosDerivSent4]
```

Now, let's unpack Source Code 4.3. `landPosIM` is an instance of an `InstanceModel`[4], containing the equational definition component (`landPosRC`), and meta-level information about the theory, including constraint ranges, notes, and a derivation[5].

To transform Source Code 4.3 into Source Code 4.2, *interpretation*[6] occurs on `landPosRC` (Source Code 4.4).

Source Code 4.4: Original relToQD

```
-- 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
```

[4]Drasil's encoding of the "Instance Models."
[5]Mostly omitted here for the sake of conserving space.
[6]More accurately, this interpretation is "domain-specific interpretation" [5].

```
convertRel sm (BinaryOp Eq (C x) r) = ec (symbResolve sm x) r
convertRel _ _ = error "Conversion failed"
```

Source Code 4.4 shows the definition of `relToQD`, a function that attempts to convert arbitrary relations into coherent "quantity definitions"[7] (`QDefinition`), which Drasil's "SRS to code" generator relies on. `landPosRC` is an instance of a `RelationConcept` (Source Code 4.5), a coupled mathematical relation (encoded using `Relation`s[8]), natural language description of the relation, and descriptive name.

Source Code 4.5: Original RelationConcept Definition

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

`relToQD` is used on *all* of the captured `InstanceModel`s of a case study to find code it can generate. However, these `InstanceModel`s carry arbitrary `Relation`s, which are arbitrary mathematical relations. As such, an issue arises: since `relToQD` only works with "variable defining" relations[9], we're limited to only one *kind* of theory. Even with this restriction, we're further limited to only a specific formation of that theory (e.g., relations of the form $x = f(a, b, c, \dots)$)!

### 4.1.2 Problems

In the sense that we really want to be able to use `relToQD` (or something like it) to transform arbitrary well-understood *theories* into code fragments for Drasil's code generator, it has at least 3 problems:

**P1** it only handles one theory kind: variable definitions,

**P2** and for those definitions, it requires a specific form, thereby limiting users to very specific usage, views, and transcription,

**P3** and it implicitly assumes that all inputs will be of this theory kind, or else it causes a panic.

As a result of P1, we aren't able to encode adequately all the theories we're interested in using, and want to generate representational code of. In particular, as Drasil is heavily guided by physics-focused case studies, ODEs are desired! When we want to use ODEs in the solution of a problem, extra information is required. For example, we might need

[7] Or, "variable definitions."

[8] `Relation` is Drasil's DSL encoding arbitrary mathematical relations. We will talk more about this later. For now, this is enough.

[9] In some sense, calling these relations "variable defining" is also an issue of itself. We're overloading = to mean definition.

to give Drasil (and/or developers) information about a desired approximation formula with particular "settings." Drasil does circumvent this issue for ODEs, but we would like to reconcile the half-measures and push all necessary information back in to the theory encodings.

Assuming we wanted to describe the theory of a line, there are many ways we can describe the equation: polynomial ($ax + by + c = 0$), slope-intercept form ($y = mx + b$), point-slope form ($y_1 - y_2 = m(x_1 - x_2)$), and so on. However, as a result of P2, we are forced to use the "simple" slope-intercept form, even though we are aware of other forms and may prefer to describe it in other forms.

As a result of P3, when Drasil's users are encoding theories in Drasil, they might be misled to think that any of their encoded theories is fully-understood to Drasil and actually usable as part of the generated solution.

Together, these issues arise because of a lack of sufficient *depth* and *breadth* in the contained knowledge of the theory encodings. Because the theory encodings rely on "flat" information (e.g., the relations), transforming them programmatically is challenging. Of course, if two programmers were to read this information in the SRS, they might be able to use it. However, the programmers understand the context of the theory, and are able to *recognize* (from any form of a theory) if and how it can be transcribed as a computation. Now, imagine if we wanted to So, how can we mitigate all of these issues?

Just as we may discuss the specifics of implementing any particular theory in our manual implementation of a software artifact, we assume prior learned knowledge about mathematical expressions, such as which ones we know we can somehow translate into code. To mitigate these issues, we must further capture this background knowledge because raw relations carry too little information about how to transform into coded. So, now, more concretely, we ask: how could we have avoided these individual problems?

| *To avoid ...* | *we needed ...* |
|---|---|
| P1 | knowledge about more *kinds* of theories, |
| P2 | to decompose the relations into its set of logical components and capture information about how instances can be transformed into various forms, |
| P3 | a signifier for each theory *kind* we're interested in. |

Succinctly, this means we need a system for *classifying* and capturing/exposing the *structure* of more *kinds* of theories. This also means `RelationConcept`s and `Relation`s are insufficient couriers of "theory knowledge." In particular, we should be careful to ensure that the abstractions are capable of reproducing the original knowledge we abstracted them out of[10]. In other words, we should ensure that we can re-create the original `RelationConcept`s from the newly created, structured, theories.

[10]This also should typically create opportunity for more kinds of interpretation of the abstracted knowledge too.

## 4.2 Classifying Theories

As a result of the inability to enter in complex theories not of the form $x = f(a, b, c, \dots)$ and the desire to encode theories with ODEs[11], Drasil's existing ODEs has a bit of temporary code to circumvent the "1 Theory Kind" restriction. However, we won't focus on ODEs in this work. Since we only have one kind of theory that is processed down the "normal" route (`relToQD`) along the path of code generation, we will focus on it.

Continuing to use $p_{\text{land}}$ as an example, recall Source Code 4.3, as it is similar to any one of the other theories in Drasil used for code generation (other than ODEs!). The code generator currently understands how to convert "quantity definitions" (`QDefinition`s, Source Code 4.6) into various snippets of usable code that it can use to generate a whole software artifact.

Source Code 4.6: "Quantity Definition" Definitions (QDefinitions)

```
-- | A QDefinition is a 'Quantity' with a defining expression, and a
↪  definition
data QDefinition = EC { _qua :: QuantityDict , _defn' :: Sentence,
↪  _equat :: Expr }
```

`QDefinition`s are "higher-level" versions of the existing theories. A `QDefinition` breaks up an equational theory (i.e., theories of the form $x = f(a, b, c, \dots)$ into its logical constituents:

1. a variable to be defined,
2. an expression (formula) defining the variable,
3. and a natural language explanation of the coupling.

For our current needs, this is a sufficient interpretation of "equational theories." Additionally, since all existing instance models are processed by `relToQD`, we similarly have a clear path to converting the existing `RelationConcept`s into `QDefinition`s where applicable. Specifically, to convert them, we merely need to hand-process `relToQD` in the case studies, where we normally write `RelationConcept`s[12]. For example, if we wanted to convert `landPosRC` into a `QDefinition`, we might do it as Source Code 4.7, where `E.landPosExpr` is a variable containing the expression (formula) defining the `landPos` symbol.

Source Code 4.7: Converting "landPosRC" into a "QDefinition"

[11] As Drasil's examples are physics-focused and ODEs are common in physics problems, the ability to generate software that can solve ODEs is desired.

[12] Note: since the `QDefinition`s are merely unpacked copies of the existing `RelationConcept`s used in code generation, we can also similarly re-create those `RelationConcept`s we're aiming to replace. In other words, we can also make `RelationConcept`s a "view" of `QDefinition` by re-interpreting them while connecting the formula to the defined variable with an equality.

```
landPosQD :: QDefinition
landPosQD = mkQuantDef landPos E.landPosExpr
```

However, merely replacing all `RelationConcept`s with `QDefinition`s is also limiting. We want to use more *kinds* of theories too! Thus, `ModelKinds` [13] (Source Code 4.8) is created and propagated through the various theory encodings to replace `RelationConcept`s (and `Relation`s).

Source Code 4.8: Enumerating and Classifying Theories

```
data ModelKinds = EquationalModel QDefinition
                | DEModel RelationConcept
                | OthModel RelationConcept
```

`ModelKinds` is an Algebraic DataType (ADT), enumerating over all currently known[14] *kinds* of theories in Drasil. The constructors of `ModelKinds` are intended to carry structured encodings of theory knowledge. By switching to `ModelKinds`, theories can be created using the structural information of theories, and the theories using structured encodings are easier to refine and interpret[15] than the unstructured copies. At the moment, `ModelKinds` has 3 kinds of "models" (theories) that can be created:

1. `EquationalModel`s, for theories that define a variable with some expression
2. `DEModel`s, for theories involve ODEs,
3. and `OthModel`s, for everything else.

Now, with `EquationalModel`, we may finally rebuild `landPosIM` (Source Code 4.3) using the improved structure (Source Code 4.9). At the moment, it does not change the expected generated artifacts, however, if desired, one can change output types of theories in the SRS (or code) by adding new kinds of `QDefinition` and `ModelKinds` interpretations.

Source Code 4.9: "landPosIM" Using "QDefinition"

```
landPosIM :: InstanceModel
landPosIM = imNoRefs (equationalModelN (nounPhraseSP
↪  "calculation of landing position") landPosQD)
```

[13] `ModelKinds` is based on a prototype by Dr. Jacques Carette. Shortly after implementing it, the Drasil Research Team discussed changing the name of "models" to "theories," to which, we did, but have not propagated through the Drasil codebase, pending analysis on how it might affect the SRS template [24]. Thus, "`ModelKinds`" might change to "`TheoryKinds`" or the singular "`TheoryKind`."

[14] Criteria for "currently known" being "ones this work has discussed up until this point."

[15] Such as for code generation!

```
  [qwC launSpeed $ UpFrom (Exc, exactDbl 0),
   qwC launAngle $ Bounded (Exc, exactDbl 0) (Exc, half $ sy pi_)]
  (qw landPos) [UpFrom (Exc, exactDbl 0)]
  (Just landPosDeriv) "calOfLandingDist" [angleConstraintNote,
  ↪  gravitationalAccelConstNote, landPosConsNote]

landPosQD :: QDefinition
landPosQD = mkQuantDef landPos E.landPosExpr

landPosDeriv :: Derivation
landPosDeriv = mkDerivName (phrase landPos) (weave [landPosDerivSents,
↪  map eS landPosDerivEqns])

landPosDerivSents :: [Sentence]
landPosDerivSents = [landPosDerivSent1, landPosDerivSent2,
↪  landPosDerivSent3, landPosDerivSent4]
```

Continuing, the `DEModel`s still require a bit of manual circumvention, but bringing some of the information back into the core "theories" to use in the intended flow of knowledge, should be possible [25]. Drasil also currently has many more kinds of theories, not all usable in code generation — more than just equational models! However, we won't focus on them *quite yet*, we'll get back to this Chapter 6.

Finally, as a result of incorporating `ModelKinds` in Drasil, we:

(a) add a system of classifying theories (through type information and constructors),

(b) change how users enter theories in Drasil (by making them enter in the structural components into "cookie cutter" shapes),

(c) gain information about theory capabilities (based on type information).

(d) and, ultimately, don't need `relToQD` anymore.

Now, on the note of exposing theory capabilities, we mean that we know have static type information, at the level of the whole Haskell code compilation, about how theories can be used. For example, if we previously had a function that converts `RelationConcept`s into pretty typeset boxes in LaTeX, we likely would have had similar troubles (as discussed above) in adding an alternative printing form of the theory. However, now, we might, for example, create a sub-kind of `EquationalModel`s that are for equations of lines. Then we can re-create the printing function for them, adding a parameter for the printing form. The printing function now is able to more easily make use of the logical constituents of a theory.

More important than printing to textual artifacts, we are interested in generating code. Specifically, we are interested in knowing which `EquationalModel`s are usable in code generation, based on their defining expressions.

# Chapter 5

# Expression Language Division

In this chapter, we will discuss how Drasil captures relations and general mathematical expressions, and the issues associated with the single universal language approach Drasil takes.

## 5.1 Relations

In Chapter 4, we discussed how Drasil captures theories. Specifically, we discussed how Drasil heavily relies on relations for defining them. However, we never discussed the relation encoding: `Relation`.

Source Code 5.1: Original Relation

```
type Relation = Expr
```

`Relation` (Source Code 5.1) is actually a type alias for `Expr` (Source Code 5.2), which captures general mathematical expressions. As such, `Relation`s aren't really "relations." In fact, `Relation`s can take on any mathematical expression, such as $3x + 2$, which is obviously not a "relation."

Source Code 5.2: Original Expression language

```
-- | 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:
  --   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
```

## 5.2 A Mathematical Language

`Expr` (Source Code 5.2) is an ADT[1], representing the Algebraic Syntax Tree (AST) of a mathematical expression language. It was grown following the needs of encoding the case studies in Drasil. As such, it contains a wide range of mathematical terms, including variables, numerics, derivatives, common mathematical operations, and function applica-

[1] In GADT syntax.

tions. In addition to the example expression (a relation) from Source Code 4.3, if, for example, we wanted to transcribe $3\tan(x) = y$ in Drasil, we might write Source Code 5.3.

Source Code 5.3: Pseudocode: Encoding of Example Expression

```
theExpr :: Expr
theExpr = dbl 3 $* tan (sy x) $= sy y
```

Of course, this is great! We're able to transcribe any expression contained in the language of `Expr`. Source Code 5.3 is a great example which we can confidently re-write as an equational model (as in Chapter 4):

Source Code 5.4: Pseudocode: Example Expression in a QDefinition

```
theQDef :: QDefinition
theQDef = mkQuantDef y (dbl 3 $* tan (sy x))
```

With Source Code 5.4, we can create an instance model containing an equational model as part of the solution of the greater problem. One of the important requirements of instance models is that we expect them to be unambiguously translatable into some component of the solution, up to user choices (such as choosing what "code" representation of integers and reals to use). This is where `Expr` shows that it is a bit of a double-edged sword. For example, we can re-write the expression used in Source Code 5.4 using a trigonometric integral (e.g., using $\tan(x) + C = \int \frac{1}{\cos^2(x)} dx$ [26], we obtain $y = \int \frac{3}{\cos^2(x)} dx$ using $C = 0$, Source Code 5.5), but then the transformation into a naive set of steps to follow isn't as simple.

Source Code 5.5: Pseudocode: Example Bad Expression in a QDefinition

```
theQDef :: QDefinition
theQDef = mkQuantDef y (integAll (eqSymb x) (dbl 3 $/ (cos (x) $^ int
↪  2)))
```

With Source Code 5.5, which solution to use isn't clear. Should Drasil generate something to solve the integral? Should Drasil replace the integral with the original tan? How would it recognize to convert it into tan without running into a similar problem as `relToQD` (Chapter 4)? Should we expect the target generated languages to have functionality for solving any integral we give it?

*We* shouldn't be the ones answering any of these questions[2]. Rather, we should be deferring these questions to users, giving them options for choosing answers to these

[2]We should be imposing the least possible amount of things on users!

questions. In other words, the transformation from `Expr`s to "code" is not naively *total*, which is problematic. However, these questions are also still too complex to be external to the SRS, and as such, we would need this theory to be a higher-level theory, which the user refines with information that would appear in the SRS for domain-experts to audit. In other words, we don't want integrals, like in this example, to appear in the concrete theories (instance models) that are used in code generation, at least without extra information about how to solve them embedded closely.

Additionally, `Expr` contains functionality for "actors" and "messages," concepts in the world of OO programming. While these are helpful for the translation of mathematical expressions that might use some functionality from "actors," users building an SRS should be relatively removed from the actual programming. In other words, these terms are inappropriate for usage in the SRS.

As such, we need to be able to restrict expression terms to their appropriate contexts. At the moment, we lack information about when instances of expressions are actually usable in code generation. As such, we're limited to relying on implicit assumptions about this, and causing Drasil-panics when the assumption is broken. To resolve this, we need to make the information *explicit* in Drasil, and use that explicit information appropriately. However, before we can make it explicit, we need to understand the contexts in which Drasil needs mathematical expressions and how they are used in those contexts.

Namely, there are at least three focal areas of interest in Drasil where we use mathematical expressions:

1. In the SRS for abstract theories, where we discuss abstract concepts that are ultimately used in the derivations of concrete theories,
2. In the SRS for concrete theories (such as instance models), where we expect them to be *concrete* in a sense that they are trivially usable as part of a solution to an outlined problem,
3. In generated artifacts, where we first need to convert the concrete theories into an intermediate representation of the generated artifacts and making any necessary specializations for the target language, before finally outputting them.

## 5.3 Splitting

To restrict terms to their appropriate contexts, we need to somehow know when users are using the "wrong" terms. Since Drasil is written in Haskell, we can use Haskell's type system to indicate this with relative ease. Namely, we can *split* up `Expr`. Thankfully, the three areas of interest share a subset of expressions: the mathematical expressions usable in concrete theories. As such, we would like to divide the language using this language as a base for the other two (Figure 5.1).

Figure 5.1: Mathematical Language Division

`Expr` ⇒ `Expr` ∪ `ModelExpr` ∪ `CodeExpr`

After the work, we end up with three languages:

1. `Expr`, which is the subset of the original `Expr` where all terms have definite values (e.g., literals, common unary and binary operations, symbols, and first-order function applications), and should be the language of the `QDefinition`s usable in instance models (Source Code A.6),

2. `CodeExpr`, which is expected to be related to a *total* transformation function that rewrites terms in `CodeExpr` into GOOL for final rendering in a final software artifact (Source Code A.10),

3. `ModelExpr`, which is approximately the `Expr` with nearly everything from the original `Expr`, except without code-oriented terms[3] (Source Code A.8). Notably, `ModelExpr` contains all indefinite valued terms that were cut out of `Expr`, such as derivatives, integrals, quantification, types as values (spaces), and continuous ranges for summations and products.

Now we have 3 ADTs available for different kinds of expression languages with "mathematics" as a common tie. One issue with splitting is that we now force users of the languages to make a conscious choice about which language they are using, and which language they need to import into their working files respectively. This can become quite frustrating given the amount of overlap. Thus, to alleviate the stress involved with writing out the same expression in different languages, we use a Typed Tagless Final (TTF) [6] encoding of the smart constructors to build expressions. Using `Expr` has a "base" shared language between `ModelExpr` and `CodeExpr`, we can write TTF encoding for them individually that extends on `Expr`'s functionality (Source Code A.7, Source Code A.9, Source Code A.11). The TTF encoding allows us to seamlessly write expressions in any of the languages at the same time, allowing type variability along the type constraints of any expression. The typeclasses used in the languages may be used to create constriants on the language used.

While `ModelExpr`s TTF encoding strictly contains the terms unique to `ModelExpr`, it is possible (and typical) to convert `Expr`s into `ModelExpr`s for usage in generating the SRS documents. Additionally, since many theories need to be representable as a single `Relation`, we may create a typeclass to create enforcement (Source Code 5.6). By instantiating it for various types, we are explaining how those terms can be interpreted as a `ModelExpr`.

Source Code 5.6: Express Typeclass

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

[3] As they should not be discussed in the SRS documents at all!

## 5.4 Back to Theories

Now that we've split up the expression languages, we may restrict their usage to appropriate theory contexts (in `ModelKinds` as per Chapter 4). Approximately, we can "upgrade" `ModelKinds` according to Source Code 5.7[4], creating a type argument in `ModelKinds` so that instance models can be restricted to carry "`ModelKinds Expr`" while the other theory types may carry "`ModelKinds ModelExpr`"[5]. This allows us to ensure that all expressions written in Drasil for instance models have an semantic counterpart that they can be translated into for code generation in various OO languages. Furthermore, now we may return to `ModelKinds` and continue uncovering the various *kinds* of theories currently discussed in Drasil's case studies.

Source Code 5.7: Pseudocode: Partial Upgrade To ModelKinds

```
data ModelKinds e where
    EquationalModel :: QDefinition e -> ModelKinds e
    DEModel         :: RelationConcept -> ModelKinds e
    OthModel        :: RelationConcept -> ModelKinds ModelExpr
```

[4]Note that we also need to slightly update `QDefinition`s: Source Code A.12 to create a type parameter for the used expression type.

[5]However, using `Expr` and `ModelExpr` is not quite accurately representative of "usable in code" versus "not," which we are going to fix [27].

# Chapter 6

# More Theory Kinds

In this chapter, we return to examining more of the existing theories in Drasil, and create alternative, structured versions of them to create further opportunities for domain-specific interpretation.

## 6.1 Remaining Theories

As of writing, there are mainly two kinds of theories that are used in code generation: the equational model and the ODE models. However, they aren't the only kinds of theories encoded in Drasil. There are at least two other re-occuring kinds of theories. One of them is a theory that shows multiple ways to define a quantity. For example, Figure 6.1 is an example from the DblPendulum case study. Meanwhile, the other theory imposes constraints on a system. For example, Figure 6.2 is an example from the SSP case study.

Figure 6.1 shows an abstract "force" variable, $\mathbf{F}$, which be defined by at least two different expressions: $m\mathbf{a}$ or $-\mathbf{T}_1 \sin(\theta_1) + \mathbf{T}_2 \sin(\theta_2)$. This theory, in particular, is used abstractly, providing reasoning for building different ways to define $\mathbf{F}$, but also creating opportunity for further specialization in other theories or systems. For example, this theory is used as part of the derivation of another theory (Figure A.1).

Meanwhile, Figure 6.2 shows a theory that explains that the conditions under which a body may be considered to be in static equilibrium. This theory was intended to be used as part of constricting other theories. For example, it was used as part of the development of another theory (Figure A.2).

Finally, with these theories in mind, we may start to add structured containers to replace the existing `RelationConcept`s and `Relation`s, currently used to build them, in hopes that we can eventually learn how we really want to use these theories.

## 6.2 "Classify All The Theories"

### 6.2.1 Equational Realms

Equational realms represent "realms" [28], sets of unique axioms that are equivalently interpretable, focused on different ways to define a particular variable. They may be specialized to become equational models. `EquationalRealm`s represent "equational realm"

Figure 6.1: Multiple Ways to Define a Variable

| | |
|---|---|
| **Refname** | **GD:xForce1** |
| **Label** | Horizontal force on the first object |
| **Units** | $N$ |
| **Equation** | $\mathbf{F} = m\mathbf{a} = -\mathbf{T}_1 \sin(\theta_1) + \mathbf{T}_2 \sin(\theta_2)$ |
| **Description** | $\boldsymbol{F}$ is the force ($N$)<br>$m$ is the mass ($kg$)<br>$\boldsymbol{a}$ is the acceleration ($\frac{m}{s^2}$)<br>$\boldsymbol{T_1}$ is the the tension of the first object ($N$)<br>$\theta_1$ is the the angle of the first rod ($rad$)<br>$\boldsymbol{T_2}$ is the the tension of the second object ($N$)<br>$\theta_2$ is the the angle of the second rod ($rad$) |
| **Source** | -- |
| **RefBy** | IM:calOfAngularAcceleration2 |

Figure 6.2: Constraints on a System

| | |
|---|---|
| **Refname** | **TM:equilibrium** |
| **Label** | Equilibrium |
| **Equation** | $\sum F_x = 0$<br>$\sum F_y = 0$<br>$\sum M = 0$ |
| **Description** | $F_x$ is the $x$-coordinate of the force ($N$)<br>$F_y$ is the $y$-coordinate of the force ($N$)<br>$M$ is the moment ($Nm$) |
| **Notes** | For a body in static equilibrium, the net forces and moments acting on the body will cancel out. Assuming a 2D problem (A:Effective-Norm-Stress-Large), the $x$-coordinate of the force $F_x$ and $y$-coordinate of the force $F_y$ will be equal to $0$. All forces and their distance from the chosen point of rotation will create a net moment equal to $0$. |
| **Source** | fredlund1977 |
| **RefBy** | GD:normForcEq, GD:momentEql, and GD:bsShrFEq |

theories in Drasil and are effectively `MultiDefn`s. For example, we may define a theory with multiple ways to define the horizontal force on an object: Source Code 6.1.

Source Code 6.1: Example of Equational Realm: Horizontal Force

```
-------------------------------------------------
-- Horizontal force acting on the first object --
-------------------------------------------------
xForceGD_1 :: GenDefn
xForceGD_1 = gdNoRefs (equationalRealmU "xForce1" xForceMD_1)
        (getUnit force) (Just xForceDeriv_1) "xForce1" []

xForceMD_1 :: MultiDefn ModelExpr
xForceMD_1 = mkMultiDefnForQuant quant EmptyS defns
    where quant = mkQuant' "force" (horizontalForce `onThe` firstObject)
                    Nothing Real (symbol force) (getUnit force)
          defns = NE.fromList [
                    mkDefiningExpr "xForceWithMass1"
                      [] EmptyS $ express $ forceGQD ^. defnExpr,
                    mkDefiningExpr "xForceWithAngle1"
                      [] EmptyS E.xForceWithAngle_1]

xForceDeriv_1 :: Derivation
xForceDeriv_1 = mkDerivName (phraseNP (force `onThe` firstObject)) [eS'
↪  xForceMD_1]
```

Source Code 6.2: "MultiDefinitions" (MultiDefn) Definition

```
-- | 'MultiDefn's are QDefinition factories, used for showing one or more
↪  ways
--   we can define a QDefinition.
data MultiDefn e = MultiDefn{
  -- | UID
  _rUid :: UID,
  -- | Underlying quantity it defines.
  _qd :: QuantityDict,
  -- | Explanation of the different ways we can define a quantity.
  _rDesc :: Sentence,
  -- | All possible ways we can define the related quantity.
  _rvs :: NE.NonEmpty (DefiningExpr e)
}
```

Source Code 6.3: Definition Possibility

```
-- | '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
}
```

### 6.2.2 Equational Constraints

"Equational constraints" are theories that assert certain properties over other theories. They use `ConstraintSet`s under the hood (Source Code 6.5) to hold a list of relations for assertion.

Source Code 6.4: Example of Equational Constraints: Equilibrium

```
equilibrium :: TheoryModel
equilibrium = tm (equationalConstraints' equilibriumCS)
  [qw fx] ([] :: [ConceptChunk])
  [] (map express equilibriumRels) [] [dRef fredlund1977] "equilibrium"
  ↪  [eqDesc]

-----------------------------------

equilibriumRels :: [ModelExpr]
equilibriumRels = map (($= int 0) . sumAll (variable "i") . sy) [fx, fy,
↪  genericM]

-- FIXME: variable "i" is a hack.  But we need to sum over something!
equilibriumCS :: ConstraintSet ModelExpr
equilibriumCS = mkConstraintSet
  (dccWDS "equilibriumCS" (nounPhraseSP "equilibrium") eqDesc) $
  NE.fromList equilibriumRels
-- makeRC "equilibriumRC" (nounPhraseSP "equilibrium") eqDesc eqRel
```

```
eqDesc :: Sentence
eqDesc = foldlSent [S "For a body in static equilibrium, the net",
  pluralNP (force `and_PP` genericM) +:+. S
   ↪ "acting on the body will cancel out",
  S "Assuming a 2D problem", sParen (refS assumpENSL) `sC` S "the",
   ↪ getTandS fx `S.and_`
  getTandS fy, S "will be equal to" +:+. eS (exactDbl 0), S "All", plural
   ↪ force,
  S "and their", phrase distance, S "from the chosen point of rotation",
  S "will create a net", phrase genericM, S "equal to" +:+ eS (exactDbl
   ↪ 0)]
```

Source Code 6.5: ConstraintSet Definition

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

### 6.2.3 Differential Equations

The capture of differential equations in Drasil is an active area of research. Dong Chen continued work here, creating `NewDEModel` [25] (a new constructor in `ModelKinds`) to start capturing information about linear ODE systems. `DEModel` is left as a temporary carriage for the remaining theories to be similarly analyzed and re-built with a deeper depth of knowledge capture, so that we can make better use of the information in them. Thanks to Dong's work, Drasil is now able to generate software for the DblPendulum case study in Java, Python, C/C++, and C# [25]. As such, this research already has some success in enabling more theories to be encoded in Drasil and appropriately used for various purposes.

### 6.2.4 Theories Left Undiscussed

While we have analyzed a few theories and how they're used, there are still many theories left undiscussed[1]. Following Drasil's methodology, they will only be analyzed and captured as necessary. Notably, `ModelKinds` still contains `OthModel`, meaning that there still exist theories in Drasil, which are used in justifications, of which we haven't yet decided how we want to use yet[2].

[1] `ModelKinds` is an incomplete enumeration.
[2] This is "future work" for now.

Finally, after Chapter 5 and this brief structuring of some existing theories, we end with `ModelKinds` appearing as in Source Code 6.6[3].

Source Code 6.6: ModelKinds

```
-- | 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.
data ModelKind e = MK {
  _mk     :: ModelKinds e,
  _mkUID  :: UID,
  _mkTerm :: NP
```

[3]In the Source Code 6.6 definition, there are two (2) TO-DO notes that you may disregard. The first one is merely a note for analyzing "well-understood" copies of our existing ODEs, and the second one refers to models that haven't yet been fully analyzed for how they will be used (other than for display).

```
}
```

Finally, as a result of implementing `ModelKinds` (Chapter 4) and the expression language division (Chapter 5), we are now able to, in at least one way, restrict the terms we use in different kinds of theories across different contexts. Ultimately, this adds some assurance that all generated artifacts only contain relevant language in them, because we have filtered out terms by their context. However, we have yet to discuss *well-typedness* of expressions.

# Chapter 7

# Typing the Expression Language

In Chapter 5, we discussed Drasil's single mathematical expression language and split it into three variants, each specialized for their intended usage context. As a result, users don't need to worry about using terms inapplicable in various contexts. However, an issue remains with expressions: Drasil's standard mathematical language (`Expr`) remains untyped, and thus Drasil does not detect malformed expressions, allowing users to make errors that affect both the problem descriptions in the SRS and the usability of generated software. In this chapter, we will focus on mitigating expression typing issues for the most important context: concrete theories used for code generation.

## 7.1 Expressions, Instance Models, and Data Definitions

To generate code, Drasil's code generator relies on a series of concrete theories as part of its input. There are two kinds of theories that the code generator uses from the abstracted SRS template: instance models and data definitions. Data definitions are, as the name suggests, concrete symbol definitions using expressions (similar to the `EquationalModel` `ModelKind` variant, discussed in Chapter 4). On the other hand, instance models may contain data definitions or any of the other applicable `ModelKinds` variants (discussed in Chapter 6, albeit currently limited[1]). For the expressions they expose/form to be usable in code generation, the expressions must be representable in GOOL somehow[2]. Thanks to Chapter 5, `Expr` has become the expression language that currently limits mathematical expressions in both instance models and data definitions to ones with definite values that can be directly computed on pencil and paper, and in most modern programming languages. However, translatability to GOOL does not necessarily mean that all the expressions are actually coherent and usable. At the moment, it is possible to give Drasil's instance models and data definitions invalid `Expr`s and have Drasil generate code.

Source Code 7.1: Pseudocode: Ill-formed Expression Defining $p_{\text{land}}$

[1]For our purposes, it's enough to assume that they can expose any number of expressions usable in code generation.

[2]Since all code generation for Drasil currently goes through GOOL.

```
landPosQD :: SimpleQDef
landPosQD = mkQuantDef landPos (int 1 $+ str "Drasil")
```

Take Source Code 4.9 for example. The defining expression of $p_{\text{land}}$, `E.landPosExpr`, is of type `Expr` and limited to terms that can be naively translated to GOOL without external knowledge. If we were to change the defining expression to something incoherent (such as 1 + `"Drasil"`[3], Source Code 7.1), we end up with (a) an algorithm that doesn't exactly make sense, and (b) a generated software source code that does what we wanted it to do (Source Code 7.2), but which isn't actually compilable software (Source Code 7.3). Despite the example's simplicity, assuming it was derived from other theories that somehow went awry, then we should have caught the type error before we generated code.

Source Code 7.2: Pseudocode: Ill-formed Expression Defining $p_{\text{land}}$: Generated Java Code

```
/** \brief Calculates landing position: the distance from the launcher to
 ↪  the final position of the projectile (m)
    \return landing position: the distance from the launcher to the final
     ↪  position of the projectile (m)
*/
public static double func_p_land() {
    return 1 + "Drasil";
}
```

Source Code 7.3: Pseudocode: Ill-formed Expression Defining $p_{\text{land}}$: Generated Java Code Compilation Error

```
Projectile/Projectile.java:52: error: incompatible types: String cannot
 ↪  be converted to double
        return 1 + "Drasil";
                   ^
1 error
```

Source Code 7.3 provides us with a meaningful error message: "String cannot be converted to double." To fix this issue, we would need to change the string to something valid, a number at least because Java also performs a secret cast, interpreting the 1 as a double, despite our encoding having it be an integer. All code Drasil generates should be reliably compilable and usable, and all errors should be noted as Drasil is used. The

[3]The sum of the number 1 and string "Drasil".

issue here lies in that the initial expression was incoherent (and the Java code was *ill-typed*). So, now, how can we avoid this situation altogether? We need all expressions to be *well-typed*. However, Drasil doesn't have a working understanding of what it means for expressions to be well-typed.

Ultimately, the issue lies in that `Expr` is not type-safe [29] and that we don't capture any knowledge about what it means for expressions to be *well-typed*. In other words, `Expr` is not closed under *preservation*[4] and *progress*[5] [29], which means that during evaluation[6] of expressions, illegal instructions and type mismatches are possible. Thus, we must make `Expr` type-safe.

## 7.2 Type-Safe Expressions

To make `Expr` type-safe, we must build a type system for it so that we can obtain information about which `Expr` terms are coherent/valid or not. A system of typing rules that dictate what well-typed expressions will allow us to determine which expressions are well-typed.

### 7.2.1 Example: A "Simple" Language

For example, if we had a small "simple" language, $\mathbb{L}$, that contains integer and boolean values, with functionality for addition, "less than" comparison, conjunction, and if-then-else (ternary operators)[7], we might write the syntax inductively, as follows[8]:

| | | | |
|---|---|---|---|
| $\mathbb{L}(l)$ | ::= | $n$ | Integers (where $n$ is any integer) |
| | \| | `true` | True |
| | \| | `false` | False |
| | \| | $l_1$ `+` $l_2$ | Addition |
| | \| | $l_1$ `<` $l_2$ | "Less than" comparison |
| | \| | $l_1 \wedge l_2$ | Conjunction |
| | \| | `if` $l_1$ `then` $l_2$ `else` $l_3$ | if-then-else (ternary "if") |

Now, we can form expressions, such as:

$$10 \tag{7.1}$$

$$23 + (400 \texttt{ + } 4000) \tag{7.2}$$

$$\texttt{if true then } 95 \texttt{ else } 96 \tag{7.3}$$

[4] *Preservation* is a rule that evaluation of a language should preserve typing of expressions [29].

[5] *Progress* is a rule that says that any well-typed expression is either a value or something that can be further evaluated [29].

[6] Of course, "evaluation" here is also related to the evaluation of GOOL-generated code.

[7] Assume the definitions of the functions be total and understood/used under the conventional sense that mathematicians so often do.

[8] Note that I will be using a traditional math-oriented syntax definition here, but if we were to transcribe it in Haskell, we may find differences.

$$42 \texttt{ + false} \tag{7.4}$$

$$\texttt{if } (0 \texttt{ < false}) \texttt{ then } 1 \texttt{ else true} \tag{7.5}$$

Now, let's evaluate these expressions. Expressions (7.1) to (7.3) can be calculated with a conventional understanding of the operations, respectively, as 10, 4423, and 95. However, Expressions (7.4) and (7.5) are worrisome. Regarding Expression (7.4), we don't have any conventional sense of addition on integers with booleans, so evaluation is unclear. In Expression (7.5), we have two issues: we're trying to relate an integer with a boolean value, and our if-then-else construction returns differently-typed values depending on the condition.

Other than modifying the syntax into a convoluted mess to avoid issues like these (which won't be easy, and might not be possible), we look to type systems to make all expressions of $\mathbb{L}$ sound. The key is in understanding that there are different "kinds" (*types*) of values (*terms*), and forming a system of figuring out which ones are valid (well-typed) or not (ill-typed). Thus, first, we must analyze and capture our universe of types of $\mathbb{L}$, $\tau$:

$$\begin{array}{rcll} \mathrm{T}(\tau) & ::= & \mathbb{B} & \text{Booleans} \\ & | & \mathbb{Z} & \text{Integers} \end{array}$$

Note that we are restricting the numeric-related operations to strictly integers. The restriction is only there for simplification of numerics. $\tau$ is an enumeration of all permissible *types* of *terms* we can have in $\mathbb{L}$.

Next, we need to add the typing rules. They will restrict our syntax to only those constructions which are semantically valid. We will do so using inference judgments, as follows[9]:

$$\frac{}{n : \mathbb{Z}} \qquad \text{(where } n \text{ is any integer.)} \tag{7.6}$$

$$\frac{}{\texttt{true} : \mathbb{B}} \qquad \text{True} \tag{7.7}$$

$$\frac{}{\texttt{false} : \mathbb{B}} \qquad \text{False} \tag{7.8}$$

$$\frac{a : \mathbb{Z} \quad b : \mathbb{Z}}{(a \texttt{ + } b) : \mathbb{Z}} \qquad \text{Addition} \tag{7.9}$$

$$\frac{a : \mathbb{Z} \quad b : \mathbb{Z}}{(a \texttt{ < } b) : \mathbb{B}} \qquad \text{"Less than" comparison} \tag{7.10}$$

$$\frac{a : \mathbb{B} \quad b : \mathbb{B}}{(a \wedge b) : \mathbb{B}} \qquad \text{Conjunction} \tag{7.11}$$

[9]Note that since there are no variables, there is no need to have a context $\Gamma$ on the left-hand side of each typing judgment.

$$\frac{b : \mathbb{B} \quad x : \tau \quad y : \tau}{(\texttt{if}\ b\ \texttt{then}\ x\ \texttt{else}\ y) : \tau} \qquad \text{if-then-else (ternary "if")} \tag{7.12}$$

So as long as we follow these typing rules while we build our expressions, when we try to evaluate any of these expressions, we should not arrive at invalid expressions where evaluation cannot be completed. In other words, so as long as expressions conform to these typing rules, the language is type-safe: preservation and progress hold. As such, we would like to build a similar system for `Expr`[10].

Initially, when we sought to implement a type system for `Expr`, we tried to piggyback on Haskells type system using type parameters in `Expr`, GADTs, and TTF. We were hopeful that we would be able to defer manually checking our desired type rules to the type signatures of expressions, having GHC provide users with elegant error messages and feedback about typing issues. While we had success with typing `Expr`, the implementation didn't fit in well with the rest of Drasil. Adding it relied on us adding type parameters to `QuantityDict`s, having users manually type in the type in the types of variables in the declarations of quantities. It also required us to use the "data kinds" Haskell extension [30] leading to difficulties with handling and flexibility of types. Additionally, we weren't able to adequately analyze and work with the types of expressions as needed. Thus, we decided rolling our own type system would be beneficial, and that we would manually check expressions as needed.

Now the question arises: how should we build it? One of the issues with the traditional declaration-style is that expressions made without explicit type annotations might have ambiguous types should we try to infer one [31]. While we don't expect any of these scenarios to occur with `Expr`, we might if we also added type-checking to `ModelExpr`, where we would like to have higher-order functions and function abstractions. Thus, we decided to build the type-checker with a modern, future-proof scheme that will scale against our expected needs [32]: with *bidirectional type-checking*.

### 7.2.2 Bidirectional Type Rules

Bidirectional typing rules break the declarative typing judgments into two typing judgments: a *check* judgment and an *inference* judgment [33]. The checking judgment, $\Gamma \vdash e \Leftarrow \tau$, says that the expression $e$ can be *checked* to be of the type $\tau$ within the context $\Gamma$. The inference judgment, $\Gamma \vdash e \Rightarrow \tau$, says that within the context $\Gamma$, we can *infer* the type of expression $e$, $\tau$. Checking and inferencing may be mutually recursive — this is helpful for type checking in cases where expressions are ambiguous and may take on many shapes (i.e., may be polymorphic) [31].

## 7.3 Typing the Expression Language

With our goal in mind and our tools in hand, to add type-checking to `Expr`, we need to: (i) capture the type universe its terms belong to, and (ii) build a system of bidirectional typing rules, and a corresponding type-checker.

[10] To begin, we will have `Expr` typed as that is the main entry for user-error, but we will eventually also add typing to the other expression languages in Drasil.

### 7.3.1 Type Universe

Thankfully, there already exists an unused type universe, `Space` (Source Code A.13), in Drasil that we can model our type universe after and potentially start using as intended.

| | | | |
|---|---|---|---|
| $\texttt{Space}(s)$ | $::=$ | $\mathbb{Z}$ | Integers |
| | $\mid$ | $\mathbb{Q}$ | Rationals |
| | $\mid$ | $\mathbb{R}$ | Reals |
| | $\mid$ | $\mathbb{N}$ | Naturals |
| | $\mid$ | $\mathbb{B}$ | Booleans |
| | $\mid$ | $\texttt{Ch}$ | Characters |
| | $\mid$ | $\texttt{String}$ | Strings |
| | $\mid$ | $\texttt{Vector}(s)$ | Vectors |
| | $\mid$ | $\texttt{Matrix}(s)$ | Matrices |
| | $\mid$ | $s_1 \times s_2 \times \ldots \times s_n \to s$ | Functions |

### 7.3.2 Type Rules

We will base our typing rules on the existing `Expr` language syntax, and focus on enriching it, not extending it for extra syntax. By applying these typing rules, we will catch any existing typing issues in the case studies.

Let $isNum(s) = s \in \{\mathbb{Z}, \mathbb{Q}, \mathbb{R}, \mathbb{N}\}$. The following inference rules make up our bidirectional type-checker's rules:

$$\frac{v : s \in \Gamma}{\Gamma \vdash v \Rightarrow s}\ \textsc{Symbols}$$

$$\frac{l \text{ is a literal of type } s}{\Gamma \vdash l \Rightarrow s}\ \textsc{Literals}$$

$$\frac{\oplus \in \{+, \cdot\} \qquad isNum(s)}{\Gamma \vdash e_1 \Rightarrow s \qquad \Gamma \vdash e_2 \Rightarrow s \qquad \ldots \qquad \Gamma \vdash e_n \Rightarrow s}{\Gamma \vdash (e_1 \oplus e_2 \oplus \ldots \oplus e_n) \Rightarrow s}\ \textsc{Assoc. Arith. Ops}$$

$$\frac{\oplus \in \{\wedge, \vee\}}{\Gamma \vdash e_1 \Rightarrow \mathbb{B} \qquad \Gamma \vdash e_2 \Rightarrow \mathbb{B} \qquad \ldots \qquad \Gamma \vdash e_n \Rightarrow \mathbb{B}}{\Gamma \vdash (e_1 \oplus e_2 \oplus \ldots \oplus e_n) \Rightarrow \mathbb{B}}\ \textsc{Assoc. Bool. Ops}$$

$$\frac{f : (s_1 \times s_2 \times \ldots \times s_n) \to s \in \Gamma}{\Gamma \vdash e_1 \Rightarrow s_1 \qquad \Gamma \vdash e_2 \Rightarrow s_2 \qquad \ldots \qquad \Gamma \vdash e_n \Rightarrow s_n}{\Gamma \vdash f(e_1, e_2, \ldots, e_n) \Rightarrow s}\ \textsc{Fun. App.}$$

$$\frac{\begin{array}{cc} \Gamma \vdash e_1 \Rightarrow s & \Gamma \vdash c_1 \Rightarrow \mathbb{B} \\ \Gamma \vdash e_2 \Rightarrow s & \Gamma \vdash c_2 \Rightarrow \mathbb{B} \\ & \vdots \\ \Gamma \vdash e_n \Rightarrow s & \Gamma \vdash c_n \Rightarrow \mathbb{B} \end{array}}{\Gamma \vdash \texttt{Cases}(e_1, \ldots, e_n, c_1, \ldots, c_n) \Rightarrow s}\ \textsc{Cases}$$

$$\frac{\begin{array}{cccc}\Gamma \vdash e_{11} \Rightarrow s & \Gamma \vdash e_{12} \Rightarrow s & \dots & \Gamma \vdash e_{1n} \Rightarrow s \\ \Gamma \vdash e_{21} \Rightarrow s & \Gamma \vdash e_{22} \Rightarrow s & \dots & \Gamma \vdash e_{2n} \Rightarrow s \\ & \vdots & & \\ \Gamma \vdash e_{m1} \Rightarrow s & \Gamma \vdash e_{m2} \Rightarrow s & \dots & \Gamma \vdash e_{mn} \Rightarrow s\end{array}}{Matrix(e_{11}, \dots, e_{mn}) \Rightarrow \texttt{Matrix}(m, n, s)} \textsc{Matrix}$$

$$\frac{isNum(s) \qquad s \neq \mathbb{N} \qquad \Gamma \vdash e \Rightarrow s}{\Gamma \vdash |e| \Rightarrow s} \textsc{Abs.}$$

$$\frac{isNum(s) \qquad s \neq \mathbb{N} \qquad \Gamma \vdash e \Rightarrow s}{\Gamma \vdash -e \Rightarrow s} \textsc{Neg.}$$

$$\frac{s \in \{\, \mathbb{R}, \mathbb{Z} \,\} \qquad \Gamma \vdash p \Rightarrow s}{\Gamma \vdash \mathtt{e}^p \Rightarrow \mathbb{R}} \textsc{Exp.}$$

$$\frac{\oplus \in \{\, \texttt{log}, \texttt{ln}, \texttt{sin}, \texttt{cos}, \texttt{tan}, \texttt{sec}, \texttt{csc}, \texttt{cot}, \texttt{arcsin}, \texttt{arccos}, \texttt{arctan}, \texttt{sqrt} \,\} \qquad \Gamma \vdash e \Rightarrow \mathbb{R}}{\Gamma \vdash \oplus e \Rightarrow \mathbb{R}} \textsc{Un. Ops}$$

$$\frac{\Gamma \vdash e \Rightarrow \mathbb{B}}{\Gamma \vdash \neg e \Rightarrow \mathbb{B}} \textsc{Logical Neg.}$$

$$\frac{isNum(s) \qquad s \neq \mathbb{N} \qquad \Gamma \vdash v \Rightarrow \texttt{Vector}(s)}{\Gamma \vdash -v \Rightarrow \texttt{Vector}(s)} \textsc{Vec. Neg.}$$

$$\frac{\Gamma \vdash v \Rightarrow \texttt{Vector}(\mathbb{R})}{\Gamma \vdash \|v\| \Rightarrow \mathbb{R}} \textsc{Vec. Norm}$$

$$\frac{\Gamma \vdash v \Rightarrow \texttt{Vector}(s)}{\Gamma \vdash \texttt{dim}(v) \Rightarrow \mathbb{Z}} \textsc{Vec. Dim.}$$

$$\frac{isNum(s) \qquad \Gamma \vdash l \Rightarrow s \qquad \Gamma \vdash r \Rightarrow s}{\Gamma \vdash \dfrac{l}{r} \Rightarrow s} \textsc{Frac. Bin. Op}$$

$$\frac{\begin{array}{c}isNum(s) \qquad s = r \vee (s = \mathbb{R} \wedge r = \mathbb{Z}) \\ \Gamma \vdash b \Rightarrow s \qquad \Gamma \vdash e \Rightarrow r\end{array}}{\Gamma \vdash b^e \Rightarrow s} \textsc{Powers}$$

$$\frac{isNum(s) \qquad \Gamma \vdash l \Rightarrow s \qquad \Gamma \vdash r \Rightarrow s}{\Gamma \vdash l - r \Rightarrow s} \textsc{Sub. Bin. Op}$$

$$\frac{\oplus \in \{\, \implies, \equiv \,\} \qquad \Gamma \vdash l \Rightarrow \mathbb{B} \qquad \Gamma \vdash r \Rightarrow \mathbb{B}}{\Gamma \vdash l \oplus r \Rightarrow \mathbb{B}} \textsc{Bool. Bin. Ops}$$

$$\frac{\oplus \in \{=, \neq\} \qquad \Gamma \vdash l \Rightarrow s \qquad \Gamma \vdash r \Rightarrow s}{\Gamma \vdash l \oplus r \Rightarrow \mathbb{B}} \text{ Eq. Bin. Ops}$$

$$\frac{\begin{array}{c}\oplus \in \{>, \geq, <, \leq\} \qquad \mathit{isNum}(s) \\ \Gamma \vdash l \Rightarrow s \qquad \Gamma \vdash r \Rightarrow s\end{array}}{\Gamma \vdash l \oplus r \Rightarrow \mathbb{B}} \text{ Ord. Bin. Ops}$$

$$\frac{\Gamma \vdash v \Rightarrow \texttt{Vector}(s) \qquad \Gamma \vdash i \Rightarrow \mathbb{N}}{\Gamma \vdash v\texttt{[}i\texttt{]} \Rightarrow s} \text{ Vec. Proj.}$$

$$\frac{\Gamma \vdash l \Rightarrow \texttt{Vector}(\mathbb{R}) \qquad \Gamma \vdash r \Rightarrow \texttt{Vector}(\mathbb{R})}{\Gamma \vdash l \times r \Rightarrow \texttt{Vector}(\mathbb{R})} \text{ Vec. Cross Prod.}$$

$$\frac{\mathit{isNum}(s) \qquad \Gamma \vdash l \Rightarrow \texttt{Vector}(s) \qquad \Gamma \vdash r \Rightarrow \texttt{Vector}(s)}{\Gamma \vdash l \cdot r \Rightarrow s} \text{ Vec. Dot Prod.}$$

$$\frac{\begin{array}{c}\oplus \in \{+, \cdot\} \qquad \mathit{isNum}(s) \\ \Gamma \vdash b \Rightarrow s \qquad \Gamma \vdash t \Rightarrow s \qquad \Gamma \vdash e \Rightarrow s\end{array}}{\Gamma \vdash \bigoplus_{b}^{t} e \Rightarrow s} \text{ “Big” Arith. Op.}$$

$$\frac{x : \mathbb{R} \in \Gamma \qquad \Gamma \vdash b \Rightarrow \mathbb{R}}{\Gamma \vdash x \in (b, \infty) \Rightarrow \mathbb{B} \qquad \Gamma \vdash x \in [b, \infty) \Rightarrow \mathbb{B}} \text{ “Is In” Interval (Bottom)}$$

$$\frac{x : \mathbb{R} \in \Gamma \qquad \Gamma \vdash t \Rightarrow \mathbb{R}}{\Gamma \vdash x \in (\infty, t) \Rightarrow \mathbb{B} \qquad \Gamma \vdash x \in (\infty, t] \Rightarrow \mathbb{B}} \text{ “Is In” Interval (Top)}$$

$$\frac{x : \mathbb{R} \in \Gamma \qquad \Gamma \vdash b \Rightarrow \mathbb{R} \qquad \Gamma \vdash t \Rightarrow \mathbb{R}}{\begin{array}{ll}\Gamma \vdash x \in (b, t) \Rightarrow \mathbb{B} & \Gamma \vdash x \in (b, t] \Rightarrow \mathbb{B} \\ \Gamma \vdash x \in [b, t) \Rightarrow \mathbb{B} & \Gamma \vdash x \in [b, t] \Rightarrow \mathbb{B}\end{array}} \text{ “Is In” Interval}$$

$$\frac{\Gamma \vdash e \Rightarrow s}{\Gamma \vdash e \Leftarrow s} \text{ Checked by Inference}$$

Notably, the “check” rule is purely based on type inference because `Expr` is a relatively straight-forward language, for which we can reasonably infer the type of any expression. However, for `ModelExpr`, this will not be the case. Finally, after having built our type universe and typing rules, we may implement bidirectional type checking and inferencing for `Expr` in Drasil.

### 7.3.3 Implementation

For a first-pass at adding a type-checker and expression validation in Drasil, it would be beneficial to be able to check each chunk and expression at once, displaying all typing

issues at once. In other words, it is helpful to perform type-checking retroactively as opposed to performing interactively, where we would be forced to create a compiler panic on sight of the first ill-typed expression. By viewing all typing issues at once, we will be able to rapidly view what type issues our type-checker implementation causes and revise accordingly.

Thus, we need to build out the typing context, $\Gamma$. $\Gamma$ is the set of expression terms for which we already know the type of. While $\Gamma$ might include whole expressions, the most important information we need while traversing `Expr`s is information about the types associated with `UID`s (since `Expr`s, when referring to chunks, only carry UID information, such as for symbols and function applications). Since each `UID` relates to a symbol with a unique type, we can build a map that carries everything for us.

Source Code 7.4: Typing Context, $\Gamma$

```
-- | We can only type check 'UID's within a type context relating 'UID's
 ↪  to
--   types since they don't carry any type information.
type TypingContext t = M.Map UID t
```

Now we may build our type checker. As we will want to implement type checking for any expression types (such as for `CodeExpr` and `ModelExpr` eventually) with respect to any type universe (not just `Space`), we prefer to build the bidirectional type checker with a typeclass[11], instantiated as needed. Using a typeclass with 2 parameters ($e$ and $t$) allows us to build out a bidirectional type checker with respect to any language, $e$, and type universe, $t$.

Source Code 7.5: Modelling Bidirectional Type Checking

```
-- | Build a bidirectional type checker for your expression language, e,
 ↪  with
--   respect to a specific type universe, t.
class (Eq t, Show t) => Typed e t where

  -- | Given a typing context and an expression, infer a unique type or
   ↪  explain
  --   what went awry.
  infer :: TypingContext t -> e -> Either t TypeError


  -- | Given a typing context, an expression, and an expected type, check
   ↪  if the
```

[11]Note: we needed to enable the "multi-parameter type-classes" [34] GHC language extension to create and use this type class, at times with the "flexible contexts" [35] extension too.

```
  --   expression can satisfy the expectation.
  check :: TypingContext t -> e -> t -> Either t TypeError
```

At the moment, `TypeError` is merely a type alias for a `String`, for which we manually typeset in a reasonably okay fashion. Eventually, it should be replaced with a whole "breadcrumbs"-like system that elegantly displays a traversal over an expression, showing where type-checking may have gone awry and why an expression is ill-typed.

Now we may build out the bidirectional type-checker for `Expr`: Source Code A.14. After which, all that's left is to actually use the type-checker.

While we would prefer to localize type-checking to the instant applicable chunks are created (and expressions too), we unfortunately aren't able to[12]. As expressions are formed and symbols are referenced, the whole symbol type context is missing. At the moment, Drasil gathers all chunks at once, registers them in memory, and then performs various "checks" and generation steps to produce software artifacts. To build the type context, we would need to build it manually, in a very tedious fashion, for each expression we build. However, this does not scale well. To localize type-checking without the extra work, we would need to make our chunk registration gradual and add a feature for exposing assertions that needs to be asserted/proven before registration in memory[13]. To make a compromise, we will settle for type-checking just before code and SRS generation, so that we can notify Drasil-users of any issues and stop generation. With this information in mind, we need to be able to extract all expressions and expected types, and relations from our `InstanceModel`s and `DataDefinition`s.

Source Code 7.6: Requiring Type-checking Constraint

```
-- | For all containers, c, which contain typed expressions, e, against a
--   specific type universe, t, expose all expressions and relations that
↪  need
--   to be type-checked.
class Typed e t => RequiresChecking c e t where
  -- | All things that need type checking.
  requiredChecks :: c -> [(e, t)]
```

With `RequiresChecking`, we have a typeclass for any instance of a type, $c$, which may contain any number of expressions of type $e$ that are expected to be type-checked with respect to a specific type from a type universe $t$. By instantiating this typeclass for `DataDefinition`s, we expose a single relation to be type-checked: that a specific `QDefinition` is well-formed. Interestingly, by instantiating this typeclass for `InstanceModel`s, we potentially expose more than one expression, depending on the `ModelKind` variant. Recall from Chapter 4 that any of our theories (including `InstanceModel`s) may take on any of our theory kinds (`ModelKinds`), and that some of them may work with and expose

[12] At least not without ease.

[13] We will take steps towards this in Chapter 8, but this task will largely be left for future work.

expressions (such as equational realms, Section 6.2.1) in various ways for use. Thus, we allow them to expose the expressions and relations individually, and we will type-check them similarly. For other theories, such as the new ODE model variant, `NewDEModel` (`DifferentialModel`), we may also have a solution system of equations [25] that need to be type-checked. Moving on, we may finally plug in our type-checker!

Source Code 7.7: Type-checking a System

```
typeCheckSI :: SystemInformation -> IO ()
typeCheckSI
  (SI _ _ _ _ _ _ ims dds _ _ _ _ _ _ chks _ _)
  = do
    -- build a variable context (a map of UIDs to "Space"s [types])
    let cxt = M.map (\(dict, _) -> dict ^. typ) (symbolTable chks)

    -- dump out the list of variables
    putStr "Symbol Table: "
    print $ M.toList cxt

    putStrLn "=====[ Start type checking ]====="
    let
      exprSpaceTups :: (HasUID t, RequiresChecking t Expr Space) => [t]
      ↪  -> [(UID, [(Expr, Space)])]
      exprSpaceTups = map (\t -> (t ^. uid, requiredChecks t))

    -- grab all type-check-able expressions (w.r.t. Space) from DDs and
    ↪  IMs
    let toChk = exprSpaceTups ims ++ exprSpaceTups dds

    -- split up theories by "ones that contain things to type check" vs
    ↪  "not",
    -- but in reverse
    let (notChkd, chkd) = partition (\(_, exsps) -> null exsps) toChk

    -- note that some theories didn't expose anything to type-check
    mapM_
      (\(t, _) -> putStrLn $ "WARNING: `" ++ show t ++
      ↪  "` does not expose any expressions to type check.")
      notChkd

    -- type check them
    let chkdd = map (second (map (uncurry (check cxt)))) chkd

    -- format 'ok' messages and 'type error' messages, as applicable
```

```
    let formattedChkd :: [Either [Char] ([Char], [Either Space
    ↪  TypeError])]
        formattedChkd = map
                          (\(t, tcs) -> if any isRight tcs
                            then Right ("`" ++ show t ++
                            ↪  "` exposes ill-typed expressions!",
                            ↪  filter isRight tcs)
                            else Left $ "`" ++ show t ++ "` OK!")
                          chkdd

    mapM_ (either
            putStrLn
            (\(tMsg, tcs) -> do
              putStrLn tMsg
              mapM_ (\(Right s) -> do
                putStr "  - ERROR: "
                putStrLn $ temporaryIndent "  " s) tcs
              )
      ) formattedChkd
    putStrLn "=====[ Finished type checking ]====="
```

By using this function before our code generation step, we are able to type-check all relations and expressions relevant to code generation. As of writing, we found enough (approximately 30) typing issues such that we decided it is best to keep type-checking done en masse, until we've cleared through them. Interestingly, we also found that we had temporary operations used in places where we needed similar variants which didn't exist yet. For example, using "absolute value" in a place where we needed vector norm, or using addition/multiplication when we needed vector addition/scaling. We may enable a compiler panic for our case studies later, once we've cleared through the majority of the existing found type errors. Notably, by adding type-checking to Drasil, we also bring into question the coherence of our SRS documents. By forcing the SRS abstraction to provide sufficiently coherent information to generate code, we also have an equal assurance that the SRS is sufficiently *coherent* for developers to manually build similar software artifacts.

To sum up, at this point, we:

- have bidirectional type-checking done for our concrete mathematical language with respect to a specific type universe, but creating a reusable interface to create similar type-checkers for other languages with respect to any type universe,
- have type-checking performed after all chunks are gathered and registered in memory, and
- uncovered many existing typing issues in Drasil's case studies.

Once the existing typing issues are resolved, we can enable program panics to enable a "hard" type-checking requirement for all relevant expressions (including, but not limited to, data definition and instance model theories).

# Chapter 8

# "Store All The Things"

In Drasil vernacular, as discussed in Chapter 3, we call encoded fragments of knowledge "chunks." In practice, users encode chunks using Haskells basic `data` types (example later), tagging it with a Unique Identifier (UID). UIDs are represented by the `UID` data type (a string of text, Source Code A.2). For example, Source Code 3.1 has the manually defined UID: *tolLoad*.

More importantly, all chunks also have a specific *type* they belong to. For example, Source Code 3.1 is a `QuantityDict` (Source Code 3.2). Drasil leverages Haskells type system to create a system of reasoning about chunks. As such, when we create instances of chunks, they are "typed" with a single fully monomorphic type signature. There is no hard criteria yet for knowing whether `data` types are chunks or not, but we do have basic requirements: they must have a UID, they must expose, in some way, what chunks they depend on, and they must be usable in some way. Commonly, we create chunks to discuss relations between other chunks. To discuss the relationships, we refer to related chunks by their UIDs rather than the whole constituent chunk to ensure that all references are the same with respect to the global set of UIDs.

We call the global set of chunks (and UIDs) the "chunk database." The chunk database (`ChunkDB`, Source Code 8.1) is built using a series of maps (Source Code A.5) that map `UID` keys to chunk data. Each map is a map from a `UID` to a specific data monomorphic data type. As such, to access a chunk by its `UID`, the associated type of the chunk must be known, even if operations we intended to use on the chunk are generic, applicable to any similar category of chunks (such as those satisfying a particular typeclass constraint set) or all chunks. Additionally, since there is one map per chunk type, all chunk types must be known and coded in the `ChunkDB` beforehand. Of course, this means the set of types that Drasil can work with is fixed and difficult to extend. For example, if we have $n$ types (with no type parameters), then there must be $n$ maps in the `ChunkDB` data type. However, for each chunk type with type parameters, each argument combination would need to also be registered. Together, this leads to an ever-growing series of maps, which scales poorly against new knowledge being "taught" to (encoded in) Drasil. Additionally, since we have a collection of maps, ensuring `UID` collisions never occur becomes tedious.

Source Code 8.1: Original Chunk Database (ChunkDB)

```
-- | Our chunk databases. Should contain all the maps we will need.
data ChunkDB = CDB { symbolTable :: SymbolMap
                   , termTable :: TermMap
                   , defTable  :: ConceptMap
                   , _unitTable :: UnitMap
                   , _traceTable :: TraceMap
                   , _refbyTable :: RefbyMap
                   , _dataDefnTable  :: DatadefnMap
                   , _insmodelTable   :: InsModelMap
                   , _gendefTable   :: GendefMap
                   , _theoryModelTable :: TheoryModelMap
                   , _conceptinsTable :: ConceptInstanceMap
                   , _sectionTable :: SectionMap
                   , _labelledcontentTable :: LabelledContentMap
                   } --TODO: Expand and add more databases
```

## 8.1 Scaling Against New Types

Fundamentally, the issue with the `ChunkDB` is that it caters to each individual type by creating a map for each type, leading to tedious busy work to make up for inflexibility.

To scale against new types, we need to make the database maps type-agnostic, merging them all together. It should contain no hard-coded references to any specific type, but describe what kinds of types are admissible (which should be any valid chunk). However, this isn't to say that we should erase all types. Rather, we should mask the type information for the purpose of storage and minimum viable usage, but ensure that the type information can be unmasked for retrieval and normal usage. Thus, the question arises: how can we achieve this?

To collapse the series of maps into a single one, we need to have a common "box" data type that can be used as the value, which carries the actual chunk data. Notably, we need a box that retains type information that we can use to unbox them to use the chunks as normally.

Source Code 8.2: Pseudocode: First Attempt at a Universal Chunk Carriage

```
data Chunk t = Chunk t
```

Now, we have a common data type to wrap our data in: `Chunk`. The type parameter exposes type information for us to be able to unpack the `Chunk` box back to its actual chunk content, so that we may use it as normal. However, this type parameter is also an issue for storage, we run into the same issue of needing to create a series of maps, one for each type:

Source Code 8.3: Pseudocode: Prospective Chunk Database

```
-- | Our chunk databases. Should contain all the maps we will need.
data ChunkDB = CDB { a   :: Map UID (Chunk A)
                   , b   :: Map UID (Chunk B)
                   , c   :: Map UID (Chunk C)
                   , ...
                   , ... :: Map UID (Chunk ...)
                   }
```

Hence, carrying type information as a parameter is not the solution. We need to have a single monomorphic data type that we can use to carry all chunk data along with their type information. The type information allows us to *safely* attempt to cast the data back to its original chunk form, before being placed into the `ChunkDB`.

Thankfully, GHC provides us with the "Existential Quantification" language feature [36][1]. "Existential Quantification" allows us to quantify over the type parameter in the `Chunk` constructor while retaining the context (the type information). In particular, this is useful for heterogeneous lists with elements, sub-typed against a set of constraints, in Haskell (which isn't normally encouraged, nor directly possible). In other words, it is useful because it allows us to ignore the type while retaining the value and a common basic functionality to all sub-typed elements through accessing the structural information through a set of constraints. With this, we can re-create `Chunk` using `ExistentialQuantification` to hide the type parameter:

Source Code 8.4: Pseudocode: Basic Chunk Box (Data Voids)

```
{-# LANGUAGE ExistentialQuantification #-}
...
data Chunk = forall a. Chunk a
```

Great! Now we can create a single map for our chunk collection.

Source Code 8.5: Pseudocode: New Chunk Database Map

```
type ChunkDB = Map UID Chunk
```

Ok, now we've created a mechanism to collapse all of our chunk maps into one, but, we've encountered an issue: we've neglected retrieval functionality. Let's see:

Source Code 8.6: Pseudocode: Broken QuantityDict Chunk Retriever

[1]We can enable it by placing a small compiler option at the top of our Haskell files (e.g., `{-## LANGUAGE ExistentialQuantification ##-}`), or, in our `package.yaml` files.

```
retrieveQD :: UID -> ChunkDB -> Maybe QuantityDict
retrieveQD u cdb = do
    (Chunk expectedQd) <- lookup u cdb
    pure expectedQd
```

`Chunk`s are currently an informational void. We have the type information accessible, but no way to use that type information to restore the chunk data to its original typed form. So, this doesn't quite work yet. For example, in Source Code 8.7, Haskell is unable to coerce the rigid parameter type $a$ into the `QuantityDict` type.

Source Code 8.7: Retriever Error

```
    • Couldn't match type ‘a’ with ‘QuantityDict’
      Expected: Maybe QuantityDict
        Actual: Maybe a
      ‘a’ is a rigid type variable bound by
        a pattern with constructor: Chunk :: forall a. a -> Chunk,
        in a pattern binding in
             a 'do' block
        at app/Main.hs:21:4-19
    • In a stmt of a 'do' block: pure expectedQd
      In the expression:
        do (Chunk expectedQd) <- lookup u cdb
           pure expectedQd
      In an equation for ‘retrieveQD’:
          retrieveQD u cdb
            = do (Chunk expectedQd) <- lookup u cdb
                 pure expectedQd
    • Relevant bindings include
        expectedQd :: a (bound at app/Main.hs:21:10)
   |
22 |   pure expectedQd
   |   ^^^^^^^^^^^^^^
```

So, how can we try to fix this? `Data.Typeable` *to the rescue!* With `Data.Typeable`, we're able to work with any piece of data given to us without being provided any information about the data directly. `Data.Typeable` allows us to create constrained types with extra functionality for casting (amongst other common reflection operations). First, we need to alter our `Chunk` data type to make the contained data satisfy the `Typeable` constraint that the `Typeable` module needs for most of its functionality. As of GHC 7.10 (for which our targeted version of 8+ is newer than), GHC automatically instantiates the `Typeable` typeclass for all types.

Source Code 8.8: Pseudocode: Examinable Chunk Box

```
{-# LANGUAGE ExistentialQuantification #-}
...
data Chunk = forall a. Typeable a => Chunk a
```

Ok, now we should be able to retrieve chunks, and cast the chunk value types as needed.

Source Code 8.9: Pseudocode: Working QuantityDict Chunk Retriever

```
retrieveQD :: UID -> ChunkDB -> Maybe QuantityDict
retrieveQD u cdb = do
    (Chunk expectedQd) <- lookup u cdb
    cast expectedQd
```

To sum up, at this point:

- We were able to mask individual chunk types by hiding them in `Chunk` boxes, allowing us to avoid discussing specific chunk types when forming our `ChunkDB`.
- `ChunkDB` is now merely a single map that allows us to easily ensure that there are no `UID` collisions.
- `ChunkDB` scales against new chunk type creation because it discusses no specific types as it only discusses types through constraints. Through this, it scales against the creation of type parameters for Haskell-level types.

Now, let's impose restrictions on the admissible chunks, to ensure that they are indeed "admissible," namely we desire the following restrictions:

1. They should contain `UID`s.
2. They should know what chunks they directly depend on.

One implicit contract we had with the chunks in the past, was that they were all expected to have `UID`s. Now, we can try to make this contract more explicit through imposing typeclass constraints. For example, Source Code 8.10 is one possible way.

Source Code 8.10: Pseudocode: UID Ownership Contract

```
class HasUID t where:
      uid :: t -> UID
```

Afterwards, we would alter our `Chunk` definition (Source Code 8.11) to add it.

Source Code 8.11: Pseudocode: Chunks with UID Constraint

```
data Chunk = forall a. (Typeable a, HasUID a) => Chunk a
```

Great! Now all chunks must "have a UID" (exposed via their `HasUID` instances), but we don't have a guarantee that they are unique and owned solely by a single instance of any chunk. This is fine for now, and is left for future work as part of future work (Section 9.3).

For the last restriction we want to impose, we want to ensure that chunks are always "understandable" in a sense that all chunks they depend exist and have already been registered in the chunk database as well. In other words, all information they depend on should have already been entered before they are entered.

Source Code 8.12: Pseudocode: Chunk Dependencies Contract

```
class HasReferences t where:
      refs :: t -> [UID]
```

Once again, we would alter our `Chunk` definition accordingly (Source Code 8.13).

Source Code 8.13: Pseudocode: Chunks with UID and Reference List Constraints

```
data Chunk = forall a. (Typeable a, HasUID a, HasReferences a) => Chunk
↪  a
```

Adding restrictions on all chunks is now also considerably simpler[2]. If we're interested, we can add a "dump to X" mechanism, where $X$ is any encoding of our choice (such as JSON), similarly. This is omitted for brevity, but we chose to add this. For now, this chunk structure will be our final specification of a chunk: an arbitrary *thing* that has a type which indicates how it's usable, a UID, and a list of *things* it references (and hence depends on) to be sufficiently "understood."

Now, we may return to discussing the `ChunkDB` structure. We often perform actions (e.g., generation, validation, etc.) on all chunks of a particular type or belonging to a set of types. Thus, *retrieval* by type is a needed feature of `ChunkDB`s. Originally, this was done by selecting the right map and using it normally. However, with the upgraded variant of chunk databases (Source Code 8.15), this is not as simple, albeit more flexible.

[2]We can also wrap up the constraints together under a single constraint by using the "constraint kinds" language extension [37] (e.g., Source Code A.17).

We can re-create this for the new `ChunkDB` as well by using `Data.Typeables` `TypeRep` signature representation of Haskell types (Source Code 8.14).

Source Code 8.14: Pseudocode: Grabbing All Chunks from the New ChunkDB

```
findAllByType :: TypeRep -> ChunkDB -> [a]
findAllByType tr cdb = catMaybes $ map (\(_, c) -> if chunkTy c == tr
↪  then Just (unChunk c) else Nothing) (M.toList cdb)

chunkTy :: Chunk -> TypeRep
chunkTy (Chunk c) = typeOf c

unChunk :: Typeable a => Chunk -> Maybe a
unChunk (Chunk c) = cast c
```

However, this is an expensive operation because it tests every chunk registered in the chunk database. Since retrieval is a commonly performed task, it will slow down Drasil. To make the `ChunkDB` more "industrial-strength," we can add an extra mechanism for caching (in various ways) to trade a bit of memory for a frequent and expensive search operation: Source Code 8.15.

Source Code 8.15: Prototyped Chunk Database

```
type ReferredBy = [UID]

type ChunkByUID = M.Map UID (Chunk, ReferredBy)

type ChunksByTypeRep = M.Map TypeRep [Chunk]

newtype ChunkDB = ChunkDB (ChunkByUID, ChunksByTypeRep)
```

Now, we have extra functionality for caching by `TypeRep`s without needing to calculate it every time we need it, saving CPU time. Additionally, we added a traceability matrix to quickly find which chunks depend on which (direct ones only, indirect ones can be calculated later if needed)[3].

Source Code 8.16: Prototyped Typed UID References

```
-- | 'TypedUIDRef' represents typed references to chunks using their
↪  'UID' and
-- type.
```

[3] i.e., a mechanism for listing which chunks depend on a specific chunk.

```haskell
newtype TypedUIDRef typ = TypedUIDRef UID

mkRef :: IsChunk t => t -> TypedUIDRef t
mkRef = TypedUIDRef . uid

typedFind :: IsChunk t => TypedUIDRef t -> ChunkDB -> Maybe t
typedFind (TypedUIDRef u) = find u

typedFindOrErr :: IsChunk t => TypedUIDRef t -> ChunkDB -> t
typedFindOrErr tu cdb = fromMaybe (error "Typed UID dereference failed.")
↪  (typedFind tu cdb)
```

With this upgraded `ChunkDB`, we can also now build typed UID references: Source Code 8.16. These are useful for chunk-level assurance that UIDs relate to something of an expected type.

`ChunkDB` is a usable core across Drasil-like projects (ones thriving on the same "knowledge-based programming" ideology). At the moment, the database prototype lies in a separate Haskell project. Work on incorporating the database in Drasil has been halted due to UID conflicts arising in the case studies. The port lies in both a separate repository (in prototype form) and a separate branch on Drasil's git repository, but can only be merged into the stable branch once we resolve some UID conflicts.

Figure 8.1: *Matryoshka Russian Dolls*, by Marco Verch [38].

The UID conflicts occur because of how they were structured. The chunks follow a "gradual building" pattern, similar to Matryoshka nesting dolls (Figure 8.1).

Source Code 8.17: Pseudocode: Example of Chunk Nesting

```
data A = A {_uid :: UID, ...}
instance HasUID A where uid = _uid

data B = B {_a :: A, ...}
instance HasUID B where uid = uid . _a

data C = C {_b :: B, ...}
instance HasUID C where uid = uid . _b
```

For example, in Source Code 8.17[4], chunks rely on a common data type for their UID. In Drasil, this is a common structure for building up chunks using smaller "inner" chunks to quickly get some basic features in a chunk, such as a name, short name, or abbreviation. In the previous generation of `ChunkDB`, UID sharing across types is allowed because there was no `UID` uniqueness checking across types. However, the new `ChunkDB`s strictly require `UID` uniqueness across chunk types because there is only one "chunk" map under it. Unfortunately, this leads to compatibility issues. As such, the structure of chunk building comes in to question. They're currently built in such a way that relies on assuming that certain chunk types exist and are related to other chunk types that share a `UID` with them. In other words, a "whole chunk" is the union of the chunks with the same `UID` spread across the other chunk maps. There are many ways that this can be resolved, all depending on the needs of "chunk building." For example, there are at least three (3) possible resolutions:

(i) forcibly/naively changing the `UID`s, creating a strict distinction between them,

(ii) forcibly/naively collapsing the chunk structure, and

(iii) restructuring the chunk structure against the chunks, possibly learning how to re-create the easy chunk building style through other features.

However, a solution has not yet been decided because this issue is largely out of scope, and requires considerable work.

[4]Which does not exist in Drasil, but is here purely for explanation purposes.

# Chapter 9

# Future Work

While this work does contribute to the Drasil research project and the ideology underpinning it, there are still many questions and concerns left unanswered. While this thesis contributes to Drasil's theories, expressions, and database, it also uncovers more questions.

## 9.1 Theories

While Chapter 4 improves upon the mathematical expression-directed method of encoding "theory knowledge," it is not without its own issues. Namely, the current implementation has two focal problems that we hope to resolve in the future: extensibility, and the questionable type parameter belonging to `ModelKind` (Source Code 6.6).

As `ModelKinds` is written using a GADT, it is difficult to extend its functionality (similar to the "expression problem" [39]). A work in progress re-design of `ModelKinds` [27], relying on GHCs `ConstraintKinds` [37] language extension, uses Haskell-level typeclasses to describe what a model satisfying the needs of being a "model kind" must provide to be treated as a "model kind". These typeclass instances may be thought of as proofs that satisfy the requirements. The primary benefit to adding extensibility to theory kinds is exactly that we become able to add any kind of *previously unknown to Drasil* theorem without needing to touch too much other code. It also allows us to restrict different types of theories to particular areas.

Continuing, the type parameter in `ModelKinds` is peculiar as 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. However, `Expr` and `ModelExpr` should not necessarily be used to denote usability of a model in code generation.

However, these two (2) issues are relatively unimportant for now. The issue of extensibility has a proof of concept solution (as discussed above) already constructed, and the issue of the type parameter is resolvable as a side effect.

Less importantly, there is also current discussion of renaming `ModelKinds` to `TheoryKind` [24], as "model" is a heavily overloaded term, and "theory" is a guaranteed discussion of abstraction, unlike "model," which is typically instantiated to some "theory."

## 9.2 Expressions

In Chapter 7, we had built a type-checker for the expression language `Expr`, and while our type errors had been sufficient for us to track down significant errors with relative ease, they are suboptimal. For example, the current type-error messages shown do not show a "breadcrumbs"-like path to the root problematic area(s) of an expression. Furthermore, we only added typing to `Expr` but not to our other related languages, `CodeExpr`, `ModelExpr`, and GOOL. In the near future, we hope to add optional typing for `ModelExpr`[1] and split up the `Space` type to accommodate the 3 variants. Finally, after having added type information to `CodeExpr`, we may begin to also add type rules to GOOL to finally assure that expression generation will only create well-typed expressions.

## 9.3 Chunks

While Chapter 8 answers questions regarding *storing* more kinds of chunks and has created a basic set of constraints that all chunks must satisfy (`HasUID` and `HasChunkRefs`), we've embedded their solutions in Haskell code rather than Drasil itself. For solutions to be included "in Drasil itself," we need to encode it such that Drasil allows users to interact with them fdynamically, without Template Haskell[40] or other things "deep" in Haskell. Similarly, we have mysterious `Typeable` usage left unknown, which should eventually be replaced with something well-understood to us. In the future, we hope to improve chunk building fundamentally, perhaps by using a DSL instead of leaning on built-in Haskell functionality. This would allow us to better analyze Drasil and its projects.

Furthermore, while we've added requirements that UIDs be unique, we haven't discussed how `UID`s should be built (e.g., automatically or manually, and how), nor ensured that `UID` uniquely refer to the chunks they were intended to link to. We desire for them to fully be *rigid designators* [41]. Perhaps these questions will naturally resolve themselves when we try to switch to using a Drasil DSL to build chunks.

As described in Chapter 8, once we resolve the issue regarding `UID` collisions, we should be able to register more of our currently underused chunks in our new `ChunkDB`s. In doing this, we will be able to perform a wider range of analysis on our necessary "knowledge" (chunks). For example, we should be able to better understand how much mental effort is needed to produce software artifacts.

Addititionally, in Section 7.3.3, we implemented type-checking for our basic mathematical expression language (`Expr`), but we added it at the *database level*, checking each chunk registered. We can abstract over what we did and make type-checking a variant of what we really did: validated properties about a chunk. Thus, we can extend the implementation for type-checking at the level of the `ChunkDB` and create general data constraints on chunks. By doing so, we will be able to categorize our chunk assertions and learn more about what was embedded in the Haskell source code.

[1] It is at times helpful to have malformed expressions for purely demonstrative purposes.

### 9.3.1 Math-specific Instances

In Chapters 4 and 7, we discussed how we can improve the reliability of mathematical language usage in Drasil in different facets. One notable facet in practice is the unit and dimension of numbers. We hope that we may, in the future, create a strong and reliable system for units and dimensions for Drasil, allowing users to discuss precision and accuracy of the generated SCS solver artifacts. Furthermore, in Chapter 7, we neglected to ensure that symbols referenced by the expression languages are all appropriate for the use-cases. For example, the `Expr` should language should only allow for reference of concrete and usable symbols, while `ModelExpr` should only allow for referencing of symbols of abstract symbols. Similarly, `CodeExpr` should only allow referencing of symbols that belong to "code" rather than to the mathematical models (for this, a switch from a `QuantityDict` is needed as well).

### 9.3.2 Database

Once the `ChunkDB` prototype finds its way into Drasil, we may focus on the ways we can manipulate them. For example, if we consider them sets we should consider implementing "union" and "intersection" operations to assist in importing and analyzing bodies of chunks. Additionally, if we consider them "tables" (as in common relational databases), we might benefit from "inner" and "outer" joins, or even using an SQL server as a backend to inputting (or handling) chunks. Using an SQL server may prove beneficial as there are already many GUI frontends for inputting RDBMS data. Furthermore, with the new database structure, "tree shaking" should be a fairly simple operation, reduced to searching for chunks that are unreachable from the universe of references of an input set of chunks. For example, for the Smith et al. SRS transformer, we may consider tree shaking against the directly captured instances of inputs and outputs to remove any unnecessary assumptions, theories, symbols, etc.

Finally, one pain-point in the case studies is inputting the transcribed chunks into the chunk database. At the moment, it requires manually inputting the data into lists (Source Code A.15), and then using the lists to form the `ChunkDB` (Source Code A.16). We could attempt to automatically register the chunks in the database, either by analyzing Haskell (meta-linguistic), or de-embedding the chunks from our Haskell code and registering on parsing them [42].

# Chapter 10

# Conclusion

In this thesis, we addressed four (4) research questions, as stated in Section 1.2:

**RQ1** *Drasil's current encoding of "theories" are essentially black boxes. We would like to use structural information present in the short list of the "kinds" of theories that show up in scientific computing. How do we codify that?*

In Chapter 4, we replaced the "black boxes" with structured versions of the same meaningful theories. In doing so, we opened up opportunities for more domain-specific interpretation of same theories, such as analysis, flexible printing, and most importantly, code generation. Chen's work [25] has anecdotally shown the success of this work. Furthermore in Chapter 6, we began dissecting Drasil's currently encoded theories, and replaced their encodings with structured variants in hopes of further usage in code generation in the future.

**RQ2** *Drasil's theory encodings rely on a single mathematical expression language, and does not expose information about applicability to different contexts. In each context (e.g., code, theories, and common arithmetic), certain terms of the expression language should be treated differently, or are simply inapplicable. How can we restrict term usage by context?*

In Chapter 5, we analyzed Drasil's single mathematical expression language, divided it according to Drasil's current needs, and created a means of using the divided variants seamlessly through creating a TTF encoding of its smart constructors. In doing this, we were able to restrict the mathematical expressions admitted in concrete theories to only those with definite values, which we can unambiguously convert into code fragments.

**RQ3** *How can we ensure that our mathematical expression language admits only valid expressions?*

In Chapter 7, we began creating a system of type-rules that our concrete mathematical expressions (`Expr`) must obey. To enforce the type-rules in Drasil, we built a bidirectional type-checker that runs on a whole SRS and reports any errors it finds. We described the bidirectional type-checking rules under a typeclass, so that we can later also add typing rules and enforcement to the other expression languages.

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

In Chapter 8, we created a chunk database structure that is capable of collecting *any* chunk that conforms to a basic set of requirements (e.g., has UID and references to required chunks, by UID).

While there is a considerable amount of future work to be done (Chapter 9), I am certain there will be an endless supply of "future work" after those tasks. Thus, with those leftover tasks in mind, I pause for reflection.

The entirety of Chapter 2 is one of the most important things I've learned by conducting this research. Another important thing I've learned is that *actual* communication of knowledge from one to others is far more complicated than it seems, and that any piece of knowledge has considerable nuance to it. Finally, by attempting to codify, and act on, our understandings of things, we're able to test just how "well-understood" [10] they truly are to us.

# Bibliography


[1] Simon Marlow et al. *Haskell 2010 Language Report*. https://www.haskell.org/onlinereport/haskell2010/. 2010 (cit. on pp. xv, 13, 18).

[2] 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 pp. xv, 18).

[3] The Drasil Team. *Drasil*. Version v0.1-alpha. 2021-02. URL: https://github.com/JacquesCarette/Drasil/tree/v0.1-alpha (cit. on pp. 1, 14, 15).

[4] 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. 1, 3, 17, 18, 20, 21).

[5] 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 pp. 2, 9, 17, 19, 23).

[6] 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 pp. 6, 10, 33).

[7] GitHub ™ and OpenAI ™. *Copilot*. 2021. URL: https://copilot.github.com/ (cit. on p. 7).

[8] 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. 8).

[9] Asif A. Siddiqi. *Beyond Earth: a chronicle of deep space exploration, 1958-2016*. Second edition. NASA SP 2018-4041. Washington, DC: National Aeronautics and Space Administration, Office of Communications, NASA History Division, 2018. 393 pp. ISBN: 978-1-62683-042-4. URL: https://www.nasa.gov/sites/default/files/atoms/files/beyond-earth-tagged.pdf (cit. on p. 8).

[10] 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. 9, 12, 67).

[11] WordPress Foundation. *WordPress*. URL: https://wordpress.org/ (cit. on p. 10).

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

[13] Django Software Foundation. *Django*. 2005. URL: https://www.djangoproject.com/ (cit. on p. 10).

[14] Taylor Otwell. *Laravel*. 2011. URL: https://laravel.com/ (cit. on p. 10).

[15] Dr. Christopher Anand Research Group. *HashedExpression*. 2020. URL: https://github.com/McMasterU/HashedExpression (cit. on p. 11).

[16] Andy Gill and Simon Marlow. *Happy. The Parser Generator for Haskell*. 1997. URL: https://www.haskell.org/happy/ (cit. on p. 11).

[17] 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. 13).

[18] 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. 16).

[19] 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. 16).

[20] 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 p. 17).

[21] Brooks MacLachlan. "A Design Language for Scientific Computing Software in Drasil". Thesis. McMaster University, 2020-07 (cit. on p. 17).

[22] 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. 17).

[23] Naveen Ganesh Muralidharan. *New example: PD Controller (#2289)*. GitHub Repository Pull Request. 2020. URL: https://github.com/JacquesCarette/Drasil/pull/2289 (cit. on p. 19).

[24] The Drasil Team. *TM versus GD versus IM versus DD Discussion (#2599)*. GitHub Repository Issues. 2021. URL: https://github.com/JacquesCarette/Drasil/issues/2599 (cit. on pp. 27, 63).

[25] Dong Chen. "Solving Higher-Order ODEs in Drasil". MA thesis. McMaster University, 2022 (cit. on pp. 28, 39, 52, 66).

[26] David Manura. *Table of Integrals*. URL: http://math2.org/math/integrals/tableof.htm (visited on 2022-12-20) (cit. on p. 31).

[27] The Drasil Team. *Alternative ModelKinds design  Display Language (#2853)*. GitHub Repository Issues. 2021. URL: https://github.com/JacquesCarette/Drasil/issues/2853 (cit. on pp. 34, 63).

[28] Jacques Carette, William M Farmer, and Michael Kohlhase. "Realms: A structure for consolidating knowledge about mathematical theories". In: *International Conference on Intelligent Computer Mathematics*. Springer. 2014, pp. 252–266 (cit. on p. 35).

[29] Robert Harper. *Practical Foundations for Programming Languages*. 2nd. Cambridge University Press, 2016 (cit. on p. 44).

[30] The Glasgow Haskell Team. *Glasgow Haskell Compiler User's Guide. Datatype promotion*. 2020-07. URL: https://downloads.haskell.org/~ghc/8.8.4/docs/html/users_guide/glasgow_exts.html#extension-DataKinds (cit. on p. 46).

[31] nLab authors. *bidirectional typechecking*. https://ncatlab.org/nlab/show/bidirectional%20typechecking. Revision 9. 2022-11 (cit. on p. 46).

[32] Jacques Carette. *Discussion with Jason Balaci*. Personal correspondence. 2022 (cit. on p. 46).

[33] David Raymond Christiansen. "Bidirectional Typing Rules: A Tutorial". In: (2013), p. 11 (cit. on p. 46).

[34] The Glasgow Haskell Team. *Glasgow Haskell Compiler User's Guide. Multi-parameter type classes*. 2020-07. URL: https://downloads.haskell.org/~ghc/8.8.4/docs/html/users_guide/glasgow_exts.html#extension-MultiParamTypeClasses (cit. on p. 50).

[35] The Glasgow Haskell Team. *Glasgow Haskell Compiler User's Guide. The superclasses of a class declaration*. 2020-07. URL: https://downloads.haskell.org/~ghc/8.8.4/docs/html/users_guide/glasgow_exts.html#extension-FlexibleContexts (cit. on p. 50).

[36] The Glasgow Haskell Team. *Glasgow Haskell Compiler User's Guide. Existentially quantified data constructors*. 2020-07. URL: https://downloads.haskell.org/~ghc/8.8.4/docs/html/users_guide/glasgow_exts.html#existentially-quantified-data-constructors (cit. on p. 56).

[37] The Glasgow Haskell Team. *Glasgow Haskell Compiler User's Guide. The Constraint Kind*. 2020-07. URL: https://downloads.haskell.org/~ghc/8.8.4/docs/html/users_guide/glasgow_exts.html#the-constraint-kind (cit. on pp. 59, 63).

[38] Marco Verch. *Matryoshka Nesting Dolls*. Licensed under Creative Commons 2.0. 2018-04. URL: https://foto.wuestenigel.com/matryoshka-russian-dolls/ (cit. on p. 61).

[39] Philip Wadler. "The Expression Problem". Message to the java-genericity electronic mailing list. 1999-11. URL: https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt (cit. on p. 63).

[40] The Glasgow Haskell Team. *Glasgow Haskell Compiler User's Guide. Template Haskell*. 2020-07. URL: https://downloads.haskell.org/~ghc/8.8.4/docs/html/users_guide/glasgow_exts.html#template-haskell (cit. on p. 64).

[41] Saul A Kripke. "Naming and necessity". In: *Semantics of natural language*. Springer, 1972, pp. 253–355 (cit. on p. 64).

[42] Jacques Carette. *Merging chunk maps into a single map via Data.Typeable (#2873).* GitHub Repository Issue Comment. 2021. URL: https://github.com/JacquesCarette/Drasil/issues/2873#issuecomment-957059049 (cit. on p. 65).

# Appendix

Source Code A.1: Pseudocode: Example Angle Equation SRS Conversion

```
\begin{minipage}{\textwidth}
\begin{tabular}{...}
\toprule \textbf{Refname} & \textbf{IM:firingAngleFormula}
\phantomsection
\label{IM:firingAngleFormula}
\\ \midrule \\
Label & Firing angle formula

\\ \midrule \\
Input & $\textit{targetDistanceFromCannon}$, $v$

\\ \midrule \\
Output & $\theta{}_c$

\\ \midrule \\
Input Constraints & --

\\ \midrule \\
Output Constraints & \begin{displaymath}
                     {\theta{}_c}\gt{}0
                     \end{displaymath}
\\ \midrule \\
Equation & \begin{displaymath}
           {\theta{}_c} = \frac{\arcsin{}
           ↪  (\frac{\textit{targetDistanceFromCannon} \cdot{}
           ↪  \mathbf{g}}{v^{2}})}{2}
           \end{displaymath}
\\ \midrule \\
Description & \begin{symbDescription}
              \item{${\theta{}_c}$ is the firing angle (${\text{rad}}$)}
              \item{${v}$ is the launch speed
               ↪  ($\frac{\text{m}}{\text{s}}$)}
```

```
                \item{$targetDistanceFromCannon$ is the distance between
                ↪  the cannon and target (${\text{m}}$)}
                \item{$\mathbf{g}$ is the gravitational acceleration
                ↪  ($\frac{\text{m}}{\text{s}^{2}}$)}
                \end{symbDescription}
\\ \midrule \\
Notes & --

\\ \midrule \\
Source & --

\\ \midrule \\
RefBy & ...

\\ \bottomrule
\end{tabular}
\end{minipage}
\paragraph{Detailed derivation of landing position:}
...
```

Source Code A.2: Original UID Definition

```
type UID = String
```

Source Code A.3: Original: Snapshot of a few of Exprs Smart Constructors

```
-- | 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.4: Original ConceptChunk

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

Source Code A.5: Original ChunkDB Type Maps

```
-- 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
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.6: Expression Language

```
-- | 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
  -- | 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.7: Expr Constructor Encoding (TTF)

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

```
  -- | 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.
  matrix :: [[r]] -> r


  -- | Applies a given function with a list of parameters.
  apply :: (HasUID f, HasSymbol f) => f -> [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.8: ModelExpr Language

```
-- | 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.
  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.9: ModelExpr Constructor Encoding (TTF)

```
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
```

Source Code A.10: CodeExpr Definition

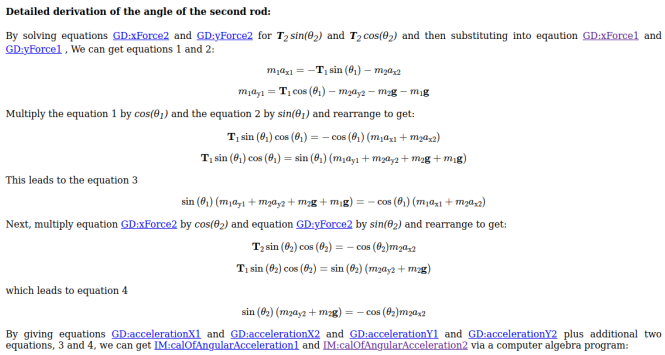
```
-- | Expression language where all terms also denote a term in GOOL
--   (i.e. translation is total and meaning preserving).
data CodeExpr where
  -- | Brings literals into the expression language.
  Lit      :: Literal -> CodeExpr


  -- | Takes an associative arithmetic operator with a list of
  ↪  expressions.
  AssocA   :: AssocArithOper -> [CodeExpr] -> CodeExpr
  -- | Takes an associative boolean operator with a list of expressions.
  AssocB   :: AssocBoolOper  -> [CodeExpr] -> CodeExpr
  -- | C stands for "Chunk", for referring to a chunk in an expression.
  --   Implicitly assumes that the chunk has a symbol.
  C        :: UID -> CodeExpr
  -- | 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 -> [CodeExpr] -> [(UID, CodeExpr)] -> CodeExpr
  -- | Actor creation given 'UID', parameters, and named parameters.
  New      :: UID -> [CodeExpr] -> [(UID, CodeExpr)] -> CodeExpr
  -- | Message an actor:
  --
  --   * 1st 'UID' is the actor,
  --   * 2nd 'UID' is the method.
  Message  :: UID -> UID -> [CodeExpr] -> [(UID, CodeExpr)] -> CodeExpr
  -- | Access a field of an actor:
  --
  --   * 1st 'UID' is the actor,
  --   * 2nd 'UID' is the field.
  Field    :: UID -> UID -> CodeExpr
  -- | For multi-case expressions, each pair represents one case.
  Case     :: Completeness -> [(CodeExpr, CodeExpr)] -> CodeExpr
  -- | Represents a matrix of expressions.
  Matrix   :: [[CodeExpr]] -> CodeExpr
```

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


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


  -- | 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 CodeExpr CodeExpr ->
  ↪  CodeExpr -> CodeExpr
  -- | The expression is an element of a space.
  -- IsIn     :: Expr -> Space -> Expr
  -- | A different kind of 'IsIn'. A 'UID' is an element of an interval.
  RealI    :: UID -> RealInterval CodeExpr CodeExpr -> CodeExpr
```

Source Code A.11: CodeExpr TTF Encoding

```
class CodeExprC r where
```

Figure A.1: Derivation of an Instance Model



**Detailed derivation of the angle of the second rod:**

By solving equations GD:xForce2 and GD:yForce2 for $\mathbf{T}_2 sin(\theta_2)$ and $\mathbf{T}_2 cos(\theta_2)$ and then substituting into eqaution GD:xForce1 and GD:yForce1 , We can get equations 1 and 2:

$$m_1 a_{x1} = -\mathbf{T}_1 \sin(\theta_1) - m_2 a_{x2}$$

$$m_1 a_{y1} = \mathbf{T}_1 \cos(\theta_1) - m_2 a_{y2} - m_2 \mathbf{g} - m_1 \mathbf{g}$$

Multiply the equation 1 by $cos(\theta_1)$ and the equation 2 by $sin(\theta_1)$ and rearrange to get:

$$\mathbf{T}_1 \sin(\theta_1)\cos(\theta_1) = -\cos(\theta_1)(m_1 a_{x1} + m_2 a_{x2})$$

$$\mathbf{T}_1 \sin(\theta_1)\cos(\theta_1) = \sin(\theta_1)(m_1 a_{y1} + m_2 a_{y2} + m_2 \mathbf{g} + m_1 \mathbf{g})$$

This leads to the equation 3

$$\sin(\theta_1)(m_1 a_{y1} + m_2 a_{y2} + m_2 \mathbf{g} + m_1 \mathbf{g}) = -\cos(\theta_1)(m_1 a_{x1} + m_2 a_{x2})$$

Next, multiply equation GD:xForce2 by $cos(\theta_2)$ and equation GD:yForce2 by $sin(\theta_2)$ and rearrange to get:

$$\mathbf{T}_2 \sin(\theta_2)\cos(\theta_2) = -\cos(\theta_2) m_2 a_{x2}$$

$$\mathbf{T}_1 \sin(\theta_2)\cos(\theta_2) = \sin(\theta_2)(m_2 a_{y2} + m_2 \mathbf{g})$$

which leads to equation 4

$$\sin(\theta_2)(m_2 a_{y2} + m_2 \mathbf{g}) = -\cos(\theta_2) m_2 a_{x2}$$

By giving equations GD:accelerationX1 and GD:accelerationX2 and GD:accelerationY1 and GD:accelerationY2 plus additional two equations, 3 and 4, we can get IM:calOfAngularAcceleration1 and IM:calOfAngularAcceleration2 via a computer algebra program:

```
  -- | Constructs a CodeExpr for actor creation (constructor call)
  new :: (Callable f, HasUID f, CodeIdea f) => f -> [r] -> r

  -- | Constructs a CodeExpr for actor creation (constructor call) that
   ↪  uses named arguments
  newWithNamedArgs :: (Callable f, HasUID f, CodeIdea f, HasUID a,
    IsArgumentName a) => f -> [r] -> [(a, r)] -> r

  -- | Constructs a CodeExpr for actor messaging (method call)
  message :: (Callable f, HasUID f, CodeIdea f, HasUID c, HasSpace c,
   ↪  CodeIdea c)
    => c -> f -> [r] -> r

  -- | Constructs a CodeExpr for actor messaging (method call) that uses
   ↪  named arguments
  msgWithNamedArgs :: (Callable f, HasUID f, CodeIdea f, HasUID c,
   ↪  HasSpace c,
    CodeIdea c, HasUID a, IsArgumentName a) => c -> f -> [r] -> [(a, r)]
     ↪  ->
    r

  -- | Constructs a CodeExpr representing the field of an actor
  field :: CodeVarChunk -> CodeVarChunk -> r
```

Source Code A.12: QDefinition Encoding

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

Figure A.2: Example of Equilibrium Usage

| | |
|---|---|
| **Refname** | **GD:normForcEq** |
| **Label** | Normal force equilibrium |
| **Units** | $\frac{N}{m}$ |
| **Equation** | $\mathbf{N}_i = (\mathbf{W}_i - \mathbf{X}_{i-1} + \mathbf{X}_i + \mathbf{U}_{g,i}\cos(\boldsymbol{\beta}_i) + \mathbf{Q}_i\cos(\boldsymbol{\omega}_i))\cos(\boldsymbol{\alpha}_i) + (-K_c\mathbf{W}_i - \mathbf{G}_i + \mathbf{G}_{i-1} - \mathbf{H}_i + \mathbf{H}_{i-1} + \mathbf{U}_{g,i}\sin(\boldsymbol{\beta}_i) + \mathbf{Q}_i\sin(\boldsymbol{\omega}_i))\sin(\boldsymbol{\alpha}_i)$ |
| **Description** | $\boldsymbol{N}$ is the normal forces ($\frac{N}{m}$)<br>$i$ is the index (Unitless)<br>$\boldsymbol{W}$ is the weights ($\frac{N}{m}$)<br>$\boldsymbol{X}$ is the interslice shear forces ($\frac{N}{m}$)<br>$\boldsymbol{U_g}$ is the surface hydrostatic forces ($\frac{N}{m}$)<br>$\boldsymbol{\beta}$ is the surface angles (°)<br>$\boldsymbol{Q}$ is the external forces ($\frac{N}{m}$)<br>$\boldsymbol{\omega}$ is the imposed load angles (°)<br>$\boldsymbol{\alpha}$ is the base angles (°)<br>$K_c$ is the seismic coefficient (Unitless)<br>$\boldsymbol{G}$ is the interslice normal forces ($\frac{N}{m}$)<br>$\boldsymbol{H}$ is the interslice normal water forces ($\frac{N}{m}$) |
| **Notes** | This equation satisfies TM:equilibrium in the normal direction. $\boldsymbol{W}$ is defined in GD:sliceWght, $\boldsymbol{U_g}$ is defined in GD:srfWtrF, $\boldsymbol{\beta}$ is defined in DD:angleB, and $\boldsymbol{\alpha}$ is defined in DD:angleA. |
| **Source** | chen2005 |
| **RefBy** | IM:fctSfty |

Source Code A.13: Original Space Definition

```
-- | Spaces
data Space =
    Integer
  | Rational
  | Real
  | Natural
  | Boolean
  | Char
  | String
  | Radians
  | Vect Space
  | Array Space
  | Actor String
  | DiscreteI [Int]  --ex. let A = {1, 2, 4, 7}
  | DiscreteD [Double]
  | DiscreteS [String] --ex. let Meal = {"breakfast", "lunch", "dinner"}
  | Void
  deriving (Eq, Show)
```

Source Code A.14: Expr's Bi-directional Type Checking Instance

```
instance Typed Expr Space where
```

```
check :: TypingContext Space -> Expr -> Space -> Either Space
↪  TypeError
check = typeCheckByInfer

infer :: TypingContext Space -> Expr -> Either Space TypeError
infer cxt (Lit lit) = infer cxt lit

infer cxt (AssocA op exs) = allOfType cxt exs sp sp
    $
    ↪  "Associative arithmetic operation expects all operands to be of the same exp
    ↪  ++ show sp ++ ")."
  where
    sp = assocArithOperToTy op

infer cxt (AssocB _ exs) = allOfType cxt exs S.Boolean S.Boolean
  $
  ↪  "Associative boolean operation expects all operands to be of the same type ("
  ↪  ++ show S.Boolean ++ ")."

infer cxt (C uid) = inferFromContext cxt uid

infer cxt (FCall uid exs) = case (inferFromContext cxt uid, map (infer
↪  cxt) exs) of
  (Left (S.Function params out), exst) -> if NE.toList params == lefts
  ↪  exst
    then Left out
    else Right $ "Function `" ++ show uid ++
    ↪  "` expects parameters of types: " ++ show params ++
    ↪  ", but received: " ++ show (lefts exst) ++ "."
  (Left s, _) -> Right $ "Function application on non-function `" ++
  ↪  show uid ++ "` (" ++ show s ++ ")."
  (Right x, _) -> Right x

infer cxt (Case _ ers)
  | null ers = Right "Case contains no expressions, no type to infer."
  | all (\(ne, _) -> infer cxt ne == eT) (tail ers) = eT
  | otherwise = Right
  ↪  "Expressions in case statement contain different types."
    where
      (fe, _) = head ers
      eT = infer cxt fe

infer cxt (Matrix exss)
  | null exss = Right "Matrix has no rows."
  | null $ head exss = Right "Matrix has no columns."
  | allRowsHaveSameColumnsAndSpace = Left $ S.Matrix rows columns t
```

```
    | otherwise = Right
     ↪ "Not all rows have the same number of columns or the same value types."
    where
        rows = length exss
        columns = if rows > 0 then length $ head exss else 0
        sss = map (map (infer cxt)) exss
        expT = head $ head sss
        allRowsHaveSameColumnsAndSpace
          = either
              (\_ -> all (\ r -> length r == columns && all (== expT) r)
               ↪ sss)
              (const False) expT
        (Left t) = expT

  infer cxt (UnaryOp uf ex) = case infer cxt ex of
    Left sp -> case uf of
      Abs -> if S.isBasicNumSpace sp && sp /= S.Natural
        then Left sp
        else Right $
         ↪ "Numeric 'absolute' value operator only applies to, non-natural, numeric t
         ↪ ++ show sp ++ "`."
      Neg -> if S.isBasicNumSpace sp && sp /= S.Natural
        then Left sp
        else Right $
         ↪ "Negation only applies to, non-natural, numeric types. Received `"
         ↪ ++ show sp ++ "`."
      Exp -> if sp == S.Real || sp == S.Integer then Left S.Real else
       ↪ Right $ show Exp ++ " only applies to reals."
      x -> if sp == S.Real
        then Left S.Real
        else Right $ show x ++ " only applies to Reals. Received `" ++
         ↪ show sp ++ "`."
    x       -> x

  infer cxt (UnaryOpB Not ex) = case infer cxt ex of
    Left S.Boolean -> Left S.Boolean
    Left sp        -> Right $ "ǹ on non-boolean operand, " ++ show sp ++
     ↪ "."
    x              -> x

  infer cxt (UnaryOpVV NegV e) = case infer cxt e of
    Left (S.Vect sp) -> if S.isBasicNumSpace sp && sp /= S.Natural
      then Left $ S.Vect sp
      else Right $
       ↪ "Vector negation only applies to, non-natural, numbered vectors. Received `
       ↪ ++ show sp ++ "`."
```

```
    Left sp -> Right $
     ↪ "Vector negation should only be applied to numeric vectors. Received `"
     ↪ ++ show sp ++ "`."
    x -> x

  infer cxt (UnaryOpVN Norm e) = case infer cxt e of
    Left (S.Vect S.Real) -> Left S.Real
    Left sp -> Right $
     ↪ "Vector norm only applies to vectors of real numbers. Received `"
     ↪ ++ show sp ++ "`."
    x -> x

  infer cxt (UnaryOpVN Dim e) = case infer cxt e of
    Left (S.Vect _) -> Left S.Integer -- FIXME: I feel like Integer would
     ↪ be more usable, but S.Natural is the 'real' expectation here
    Left sp -> Right $ "Vector 'dim' only applies to vectors. Received `"
     ↪ ++ show sp ++ "`."
    x -> x

  infer cxt (ArithBinaryOp Frac l r) = case (infer cxt l, infer cxt r) of
    (Left lt, Left rt) -> if S.isBasicNumSpace lt && lt == rt
      then Left lt
      else Right $
       ↪ "Fractions/divisions should only be applied to the same numeric typed operan
       ↪ ++ show lt ++ "` / `" ++ show rt ++ "`."
    (_       , Right e) -> Right e
    (Right e, _       ) -> Right e

  infer cxt (ArithBinaryOp Pow l r) = case (infer cxt l, infer cxt r) of
    (Left lt, Left rt) -> if S.isBasicNumSpace lt && (lt == rt || (lt ==
     ↪ S.Real && rt == S.Integer))
      then Left lt
      else Right $

         ↪ "Powers should only be applied to the same numeric type in both operands,
         ↪ ++ show lt ++ "` ^ `" ++ show rt ++ "`."
    (_       , Right x) -> Right x
    (Right x, _       ) -> Right x

  infer cxt (ArithBinaryOp Subt l r) = case (infer cxt l, infer cxt r) of
    (Left lt, Left rt) -> if S.isBasicNumSpace lt && lt == rt
      then Left lt
      else Right $
       ↪ "Both operands of a subtraction must be the same numeric type. Received `"
       ↪ ++ show lt ++ "` - `" ++ show rt ++ "`."
    (_, Right re) -> Right re
```

```
    (Right le, _) -> Right le

  infer cxt (BoolBinaryOp _ l r) = case (infer cxt l, infer cxt r) of
    (Left S.Boolean, Left S.Boolean) -> Left S.Boolean
    (Left lt, Left rt) -> Right $
    ↪ "Boolean expression contains non-boolean operand. Received `" ++
    ↪ show lt ++ "` & `" ++ show rt ++ "`."
    (_      , Right er) -> Right er
    (Right el, _      ) -> Right el

  infer cxt (EqBinaryOp _ l r) = case (infer cxt l, infer cxt r) of
    (Left lt, Left rt) -> if lt == rt
      then Left S.Boolean
      else Right $
      ↪ "Both operands of an (in)equality (=/≠) must be of the same type. Received
      ↪ ++ show lt ++ "` & `" ++ show rt ++ "`."
    (_, Right re) -> Right re
    (Right le, _) -> Right le

  infer cxt (LABinaryOp Index l n) = case (infer cxt l, infer cxt n) of
    (Left (S.Vect lt), Left nt) -> if nt == S.Integer || nt == S.Natural
    ↪ -- I guess we should only want it to be natural numbers, but
    ↪ integers or naturals is fine for now
      then Left lt
      else Right $
      ↪ "List accessor not of type Integer nor Natural, but of type `"
      ↪ ++ show nt ++ "`"
    (Left lt         , Left _)  -> Right $
    ↪ "List accessor expects a list/vector, but received `" ++ show lt
    ↪ ++ "`."
    (_               , Right e) -> Right e
    (Right e         , _      ) -> Right e

  infer cxt (OrdBinaryOp _ l r) = case (infer cxt l, infer cxt r) of
    (Left lt, Left rt) -> if S.isBasicNumSpace lt && lt == rt
      then Left S.Boolean
      else Right $
      ↪ "Both operands of a numeric comparison must be the same numeric type, got:
      ↪ ++ show lt ++ ", " ++ show rt ++ "."
    (_, Right re) -> Right re
    (Right le, _) -> Right le

  infer cxt (VVVBinaryOp Cross l r) = case (infer cxt l, infer cxt r) of
    (Left lTy, Left rTy) -> if lTy == rTy && S.isBasicNumSpace lTy && lTy
    ↪ /= S.Natural
      then Left lTy
```

```
      else Right $
       ↪ "Vector cross product expects both operands to be vectors of non-natural nu
       ↪ ++ show lTy ++ "` X `" ++ show rTy ++ "`."
    (_       , Right re) -> Right re
    (Right le, _       ) -> Right le

  infer cxt (VVNBinaryOp Dot l r) = case (infer cxt l, infer cxt r) of
    (Left lt@(S.Vect lsp), Left rt@(S.Vect rsp)) -> if lsp == rsp &&
     ↪ S.isBasicNumSpace lsp
      then Left lsp
      else Right $
       ↪ "Vector dot product expects same numeric vector types, but found `"
       ↪ ++ show lt ++ "` ů `" ++ show rt ++ "`."
    (Left lsp, Left rsp) -> Right $
     ↪ "Vector dot product expects vector operands. Received `" ++ show
     ↪ lsp ++ "` ů `" ++ show rsp ++ "`."
    (_, Right rx) -> Right rx
    (Right lx, _) -> Right lx

  infer cxt (Operator aao (S.BoundedDD _ _ bot top) body) =
    let expTy = assocArithOperToTy aao in
    case (infer cxt bot, infer cxt top, infer cxt body) of
      (Left botTy, Left topTy, Left bodyTy) -> if botTy == S.Integer
        then if topTy == S.Integer
          then if expTy == bodyTy
            then Left expTy
            else Right $ "'Big' operator range body not Integer, found: "
             ↪ ++ show bodyTy ++ "."
          else Right $ "'Big' operator range top not Integer, found: " ++
           ↪ show topTy ++ "."
        else Right $ "'Big' operator range bottom not of expected type: "
         ↪ ++ show expTy ++ ", found: " ++ show botTy ++ "."
      (_         , _         , Right x    ) -> Right x
      (_         , Right x   , _          ) -> Right x
      (Right x   , _         , _          ) -> Right x

  infer cxt (RealI uid ri) =
    case (inferFromContext cxt uid, riTy ri) of
      (Left S.Real, Left riSp) -> if riSp == S.Real
        then Left S.Boolean
        else Right $

          ↪ "Real interval expects interval bounds to be of type Real, but received
          ↪ ++ show riSp ++ "."
      (Left uidSp, _         ) -> Right $
```

```
        ↪ "Real interval expects variable to be of type Real, but received `"
        ↪ ++ show uid ++ "` of type `" ++ show uidSp ++ "`."
      (_          , Right x  ) -> Right x
      (Right x    , _        ) -> Right x
    where
      riTy :: RealInterval Expr Expr -> Either Space TypeError
      riTy (S.Bounded (_, lx) (_, rx)) = case (infer cxt lx, infer cxt
       ↪ rx) of
        (Left lt, Left rt) -> if lt == rt
          then Left lt
          else Right $

            ↪ "Bounded real interval contains mismatched types for bottom and top. E
            ↪ ++ show lt ++ "` to `" ++ show rt ++ "`."
        (_      , Right x) -> Right x
        (Right x, _      ) -> Right x
      riTy (S.UpTo (_, x)) = infer cxt x
      riTy (S.UpFrom (_, x)) = infer cxt x
```

Source Code A.15: Example of Current Chunk Gathering Into A List

```
dataDefs :: [DataDefinition]
dataDefs = [risk, hFromt, loadDF, strDisFac, nonFL, glaTyFac,
  dimLL, tolPre, tolStrDisFac, standOffDis, aspRat, eqTNTWDD,
   ↪ probOfBreak,
  calofCapacity, calofDemand]
```

Source Code A.16: Example of Current Gathering Chunk Lists Into the Database

```
symbMap :: ChunkDB
symbMap = cdb thisSymbols (map nw acronyms ++ map nw thisSymbols ++ map
 ↪ nw con
  ++ map nw con' ++ map nw terms ++ map nw doccon ++ map nw doccon' ++
   ↪ map nw educon
  ++ [nw sciCompS] ++ map nw compcon ++ map nw mathcon ++ map nw mathcon'
  ++ map nw softwarecon ++ map nw terms ++ [nw lateralLoad, nw
   ↪ materialProprty]
   ++ [nw distance, nw algorithm] ++
  map nw fundamentals ++ map nw derived ++ map nw physicalcon)
```

```
  (map cw symb ++ terms ++ Doc.srsDomains) (map unitWrapper [metre,
  ↪  second, kilogram]
  ++ map unitWrapper [pascal, newton]) GB.dataDefs iMods [] tMods concIns
  ↪  section
  labCon []
```

Source Code A.17: Chunk Constraints, Wrapped Together

```
-- | Constraint for anything that may be considered a valid chunk type.
type IsChunk a = (HasUID a, HasChunkRefs a, Typeable a)
```