Capturing Mathematical Knowledge in Drasil: the Case of Theories

Jason Balaci (balacij), Dr. Jacques Carette (carette)
Department of Computing and Software, McMaster University

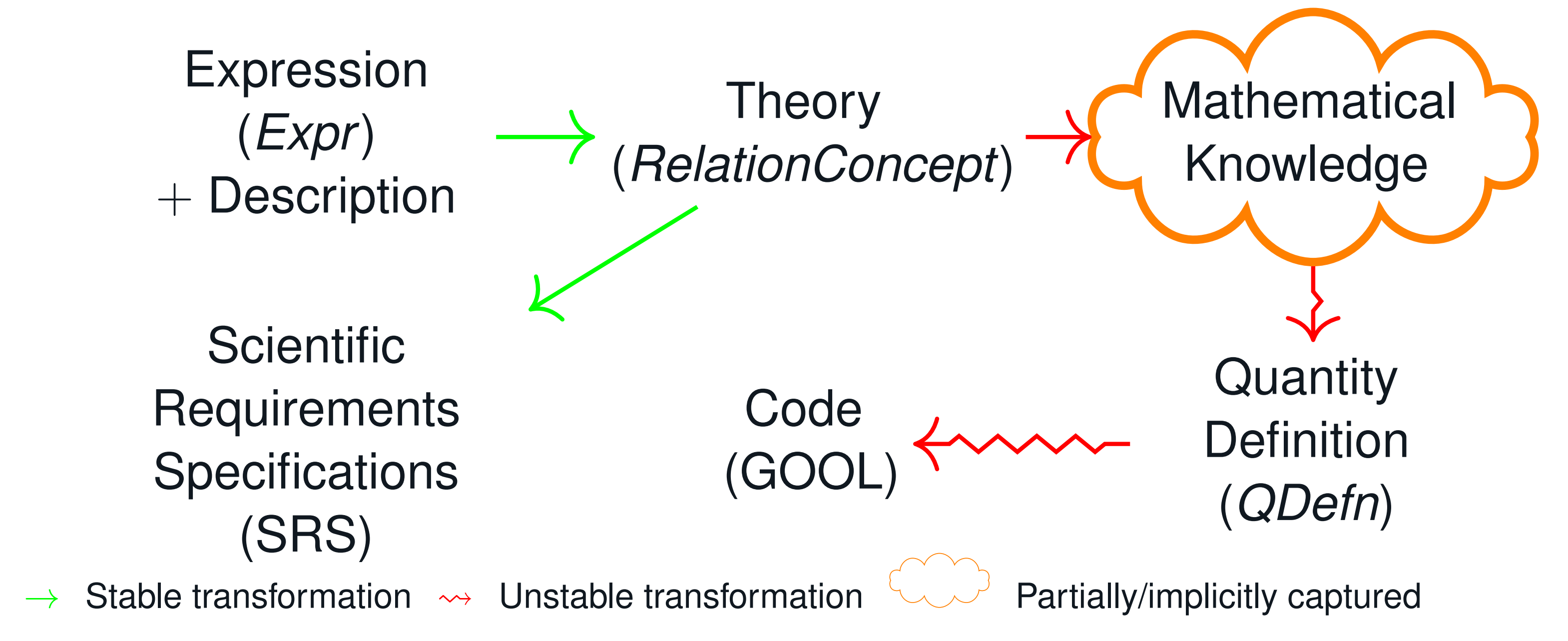
What is Drasil?

Drasil is a framework for generating families of software artifacts from a coherent knowledge base, following its mantra; “Generate All The Things!”. Drasil uses a series of variably sized Domain-Specific Languages (DSLs) to describe various fragments of knowledge that domain experts and users alike may use to piece together fragments of knowledge into a coherent “story”. Through forming some coherent “story” in a domain captured by Drasil, a representational software artifact may be generated. Drasil currently focuses on Scientific Computing Software (SCS), following Smith and Lai’s Software Requirements Specifications (SRS) template as described in [4]. Behind the scenes of the SRS, a mathematical language is used to describe various theories, and have representational software constructed via compiling to Generic Object-Oriented Language (GOOL) [2]. Through encoding knowledge in Drasil, an increase in productivity (and maintainability) in building reliable and traceable software artifacts is observed [5], specifically in SCS [3]. Drasil’s source code, case studies, and documentation studies can be found on its website; `https://jacquescarette.github.io/Drasil/`.

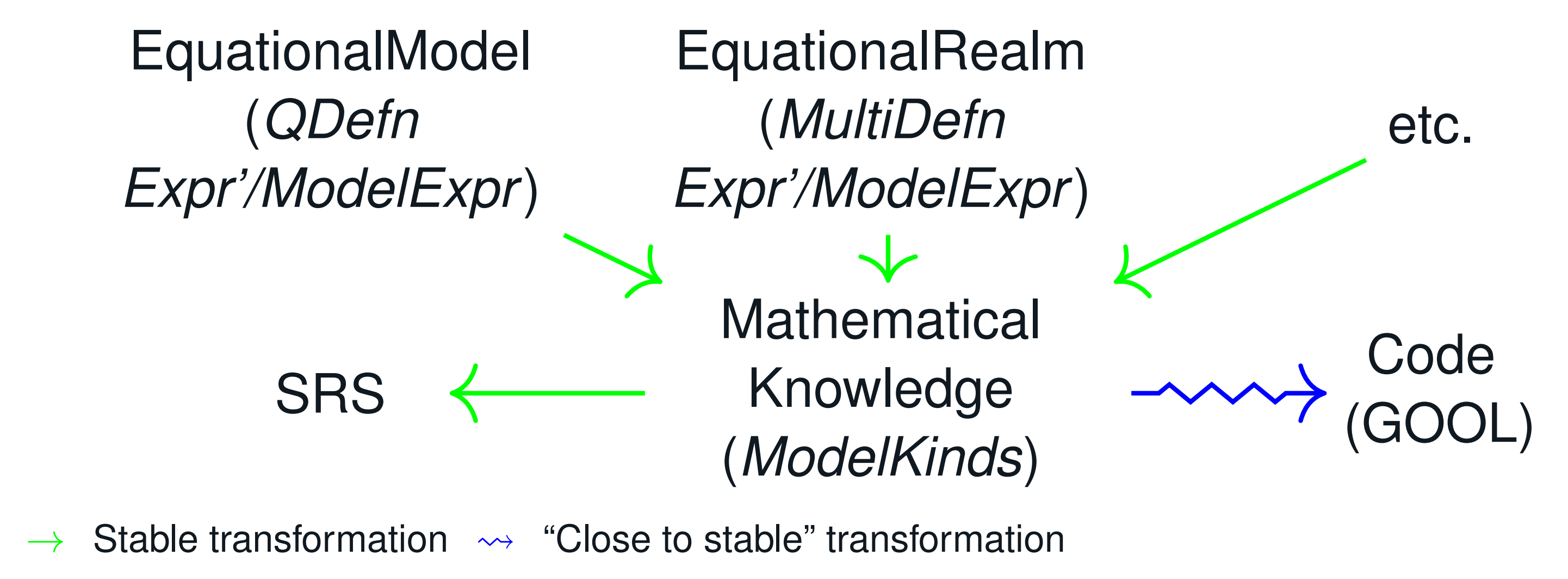
Research Motivation/Problem

- A single mathematical expression language, and a single instance of a term from the language is used to describe theories (*RelationConcept*) alongside a natural English description.
  - Expressions must be precisely written in a manner “digestible” to the code generator so that a representational software code snippet can be constructed.
  - Not all expressions have definite values and are immediately usable in all programming languages.
- Only a handful of the case studies generate code, because...
  - Code is validated, and has rigid rules we must obey, leaving little to no room for error.
  - Cognitive load of writing expressions in precise manners to accommodate the code generator would increase.
  - Understanding & usage of the expressions is weak and brittle as they don’t expose sufficient information about the models.

Mathematical Knowledge Flow



Capturing Mathematical Knowledge



What changed?

- Expression language division: *Expr* $\longrightarrow$ *Expr'* $\cup$ *ModelExpr* $\cup$ *CodeExpr*
  - *Expr'*: Restricting the language to terms “well-understood”, with a definite meaning and value.
  - *ModelExpr*: Restricting the language to terms with definite meanings, but not necessarily definite values.
  - *CodeExpr*: Restricting the language to terms that have a definite meaning and value to most general purpose programming languages, with some goodies for OOP.
- Created Typed Tagless Final (TTF) [1] encodings; Expression creation is just as easy as it ever was!
- Created a system of classifying *theories* (*ModelKinds*)
  - Increased the depth & breadth of knowledge contained.
  - First-class representation of theories, with their fundamental components flexibly usable. No more brittle cast-like conversion of mathematical expressions (low information density) to well-understood pieces of knowledge (high information density).
  - Instances of theories usable in a wide variety of ways can be statically & reliably checked for validity.
  - Creating instances of theories comes with projectional editor-like ease.
- Improving productivity, stability, and flexibility in usage
- Current theory types:
  - *EquationalModel*: $x = f(a, b, c, \ldots)$
  - *EquationalRealm*: $x = a \vee x = b \vee x = c \vee \ldots$
  - *EquationalConstraints*: $a \wedge b \wedge c \wedge \ldots$
  - *DEModel*: $\ldots \frac{dy}{dx} \ldots$

Conclusion & Future Work

- Through capturing and classifying mathematical knowledge, we earn significant gains in flexibility, usage, and handling.
- *In progress*: Adding extra type information to expressions, allowing us to add type information to GOOL, and further improve static validity rules of various mathematical constructions. This would resolve the “Close to stable” transformation blue arrow.

References

[1] 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.

[2] Jacques Carette, Brooks MacLachlan, and Spencer Smith. “GOOL: a generic object-oriented language”. In: *Proceedings of the 2020 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation.* 2020, pp. 45–51.

[3] 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.

[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.