Validating the Formalization of Theories and Algorithms of Discrete Mathematics by the Computer-Supported Checking of Finite Models

Report on Bachelor Thesis in Seminar for Computer Algebra

Alexander Brunhuemer

March 9, 2017

What is my thesis about?

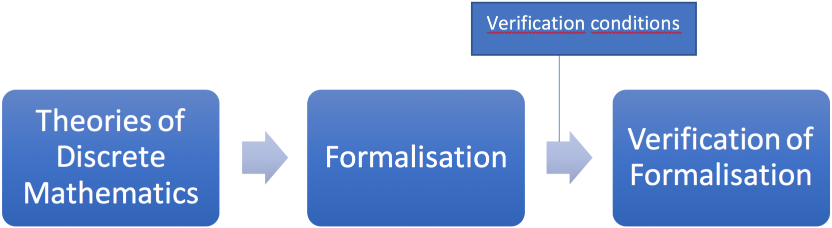
Why formalise mathematical theories?

- Machines and new technologies brought new possibilities for mathematicians
- Computers don't allow interpretations, they just do, what the programmer or user tells them to do

Verification of Formalisations

- Developers created tools to support the process of formalisation and its verification
- Only way to ensure correctness of a program, which operates over an unbounded domain, is to generate verification conditions
- Typically this demands human interaction in form of annotations (potential error source)
- So we need a way to make sure the chosen annotations are correct
- Possible solution: restrict domain of values to a finite number instead of an unbounded domain and apply model-checking
- Finite domain annotations can then be generalized to unbounded domains

Verification of Formalisations



State of the Art

Tools	Can be used for...
Coq	writing executable algorithms and extracting them into functional programs. Correctness can be verified -> programs are correct
Isabelle	the higher order logic of this generic proof assistant embeds similar functionality. Both tools aim to generate executable code from verified algorithms, but they do not validate the correctness of algorithms before verification.
SETL	is an algorithm language, very high-level language based on set theory but doesn't support formal specifications
Alloy	is a language which is completely based on relations and allows to describe structures and their relationships. As a consequence it's pretty complicated to define mathematical algorithms with it (e.g. a loop is specified in Alloy by describing the changes of the variables during an iteration of the loop).
JML	extension for Java for specification, which also allows introduction of loop invariants and other annotations. However, it struggles with the complex semantics of Java, when it comes to expressive specifications
TLA/PlusCal	TLA with the extension PlusCal allows defining mathematical algorithms in a very convenient way. Additionally it includes a model-checker, which yields an error and the complete path, when it violates properties of the algorithm. One essential disadvantage is still the missing implementation of recursive algorithms.
VDM	The language includes mathematical objects like sets and functions and therefore it's very helpful in defining mathematical algorithms. Moreover it allows to define recursive functions. Still it has its' deficiencies in defining verification conditions, since it is only possible to specify system conditions and not e.g. for individual loops.

RISCAL - What does RISCAL do?

- Supports process of verification
- Formulation of mathematical theories and high-level algorithms
- based on a type system which ensures that all variable domains are finite at any time

RISCAL - What does RISCAL do?

- RISCAL validates the meaningfulness of definitions, the truthfulness of propositions and correctness of programs automatically, by evaluation of terms and formulas and executing programs over all possible inputs
- Master thesis is in progress to generate verification conditions and verify them by SMT solvers

RISCAL - How can RISCAL support a developer?

- Supports process of verification
- Provides a very intuitive way to describe the mathematical theories and algorithms
- Allows usage of many symbols used in mathematics, which makes the code very easy to read

ASCII String	Unicode Character
Int	ℤ
Nat	ℕ
:=	:=
true	⊤
false	⊥
~	¬
/\	∧
\/	∨
=>	⇒
<=>	⇔
forall	∀
exists	∃
sum	∑
product	∏
~=	≠
<=	≤
>=	≥
*	·
times	×
{}	∅
intersect	∩
union	∪
Intersect	⋂
Union	⋃
isin	∈
subseteq	⊆
<<	⟨
>>	⟩

RISCAL - How can RISCAL support a developer?

- **Types:** With types we introduce the mathematical objects we are working on in our further specifications
- **Predicates:** are boolean-valued functions which describe, if a given property is either true or false for a given input of our types
- **Functions:** are relations between a given set of inputs to the according set of outputs
    - *Implicit:* declares, which predicates a result shall fulfil, but don't give a constructive way how to find such a result
    - *Explicit:* Explicit Functions describe a constructive way to find such a result recursively
- **Theorems:** special forms of predicates, for which we predict that all applications yield "true"
- **Procedures:** returns a value to a given input, after executing commands in sequence and evaluating the according expressions

RISCAL - How can RISCAL support a developer?

The definition of a function, predicate, theorem or procedure may also include (in form of annotations):

- preconditions (requires),
- postconditions (ensures) and
- termination measures (decreases)

RISCAL also aims to give an understanding of the connections of all these definitions mentioned before.

First Example

A little example out of the handbook to give you an idea, what I'm talking about.

Expected Results

The thesis results in a collection of formalised mathematical theories and algorithms including:

- **Specifications**
    - types, predicates, functions, theorems and procedures
    - pre-, postconditions and termination measures
- **Validation outcomes**
    - validation of theorems
    - validation of specifications
    - validation of loop annotations

Expected Results

Chosen areas from discrete mathematics:

- **Set Theory**
  - Operations on sets ($\cup, \cap, \complement, \ldots$)
  - Stating rules as theorems (associative law, distributive law,?)
  - Cartesian product
  - Cardinality
  - ...
- **Relation Theory**
  - Compositions of relations
  - Inverse relations
  - Relations of special type (reflexive, symmetrical, transitive,...)
  - ...
- **Graph Theory**
  - Subgraphs, Union,...
  - Paths and components
  - Trees
  - ...

Procedure of Specification

1. Define parameters for domain size
2. Introduce types
3. Define functions/predicates with set theory
4. Prove that the definition is equal to the implemented operator (if one exists)
5. Specify procedure and define invariants with help of defined function/predicate (or the built-in operator, because of efficiency reasons)
6. Recursive function definition with specification and termination term based on defined function/predicate

Demonstration

And here is what I specified for the Bachelor's thesis until today.

Future Work

- Finish set theory,
- go on to relation theory and graph theory
- there will be another Bachelor's thesis in the area of computer algebra