Formally verified algorithms and data structures in Coq: concepts and techniques

(Formalnie zweryfikowane algorytmy i struktury danych w Coqu: koncepcje i techniki)

Wojciech Kołowski

Praca magisterska

Promotor: dr Małgorzata Biernacka

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

Czerwiec '20 chyba że koronawirus

Abstract

We discuss how to design, implement, specify and verify functional algorithms and data structures, concentrating on formal proofs rather than asymptotic complexity or actual performance. We present concepts and techniques, both of which often rely on one key principle – the reification and representation, using Coq's powerful type system, of something which in the classical-imperative approach is intangible, like the flow of information in a proof or the shape of a function's recursion. We illustrate our approach using rich examples and case studies.

Omawiamy sposoby projektowania, implementowania, specyfikowania i weryfikowania funkcyjnych algorytmów i struktur danych, skupiając się bardziej na dowodach formalnych niż na asymptotycznej złożoności czy faktycznym czasie działania. Prezentujemy koncepcje i techniki, obie często opierające na jednej kluczowej zasadzie – reifikacji i reprezentacji, za pomocą potężnego systemu typów Coqa, czegoś co w klasycznym, imperatywnym podejściu jest nieuchwytne, jak przepływ informacji w dowodzie czy kształt rekursji funkcji. Nasze podejście bogato ilustrujemy przykładami i studiami przypadku.

Contents

1 Introduction **7**

1.1 The overarching paradigm 8

1.2 Two flavours of algorithms 9

1.3 The many-worlds interpretation of imperative algorithms 11

1.4 Formal verification as a world-collapser 14

1.5 Type theory (as a model of computation) 16

1.6 Way of the Coq 18

1.7 An ultra short literature review 21

2 A quick *tour de* sort **23**

3 A man, a plan, a canal – MSc thesis **25**

3.1 Design 25

3.2 Techniques 25

3.3 Topics 25

4 Conclusion **27**

Bibliography **29**

Chapter 1

Introduction

The title of this thesis is "Formally verified functional algorithms and data structures in Coq: concepts and techniques". In the Introduction, we take time to carefully explain what we mean by each of these phrases:

- In section 1 we look at the social context that gives rise to an interesting misconception about "algorithms and data structures".
- In section 2 we explain "functional" programming by comparing it with imperative programming.
- Section 3 is a philosophical interlude in which we give an interesting interpretation of typical algorithmic activities as living in different worlds, with links between worlds being the source of potential errors.
- In section 4 we lay out the (almost) unity of the "formally verified" algorithmic world by contrasting it with the many worlds of the previous section.
- In section 5 we describe Martin-Löf Type Theory, a formal system very close to the theoretical underpinnings of Coq, which can be seen as our model of computation.
- In section 6 we show how working "in Coq" looks like.
- In section 7 we do an ultra short literature review and discuss how our "concepts and techniques" differ from these that can be found there.

In the remaining chpaters we describe the actual concepts and techniques, using well-known and well-studied examples like quicksort, merge sort or binary heaps:

- TODO

1.1 The overarching paradigm

The Free Dictionary says [1] that an algorithm is

> A finite set of unambiguous instructions that, given some set of initial conditions, can be performed in a prescribed sequence to achieve a certain goal and that has a recognizable set of end conditions.

The purpose of this entry is to explain the concept to a lay person, but it likely sounds just about right to the imperative programmer's ear too. To a functional ear, however, talking about sequences of instructions most certainly sounds as unfunctional as it possibly could. It is no surprise then that some people wonder if it is even possible for algorithms to "exist" in a functional programming language, as exemplified by this StackOverflow question [2]. The poor soul asking this question had strongly associated algorithms with imperative languages in his head, even though functional languages have their roots in lambda calculus, a formal system invented precisely to capture what an algorithm is.

This situation is not uncommon and rather easy to explain. *Imperative* algorithms and data structures [1] form one of the oldest, biggest, most widespread and prestigious fields of theoretical computer science. They are taught to every student in every computer science programme at every university. There's a huge amount of books and textbooks, with classics such as [3] [4] known to pretty much everybody, at least by title. There's an even huger and still growing mass of research articles published in journals and conferences and I'm pretty sure there are at least some (imperative) algorithm researchers at every computer science department in existence.

But theoretical research and higher education are not the only strongholds of imperative algorithms. They are also pretty much synonymous with competitive programming, dominating most in-person and online computer science competitions like ICPC and HackerRank, respectively. They are seen as the thing that gifted high school students interested in computer science should pursue – each time said students don't win medals in the International Olympiad in Informatics or the like, there will be some journalists complaining that their country's education system is "falling behind". In many countries, like Poland,[2] if there's any high school level computer science education besides basic programming, it will be in imperative algorithms.

Imperative algorithms aren't just a mere field of study – they are more of a mindset and a culture; following Kuhn's [5] terminology, they can be said to form a paradigm. Because this paradigm is so immense, so powerful and so entrenched, we feel free to completely disregard it and devote ourselves and this thesis to studying a related field which did not yet reach the status of a paradigm – functional algorithms – focusing on proving their formal correctness.

---

[1] From now on when we write "algorithms" we will mean "algorithms and data structures"

[2] I believe the same is true for the rest of the former Eastern Bloc countries too and probably not much better in the West either.

But before we do that, we spend the rest of this chapter comparing the imperative and functional approaches to algorithms and briefly reviewing available literature on functional algorithms.

1.2 Two flavours of algorithms

The differences between the fields imperative and functional algorithms are mostly a reflection of the differences between imperative and functional programming languages and only in a small part a matter of differences in research focus.

The basic data structure in imperative languages is the array, which abstracts a contiguous block of memory holding values of a particular type. More advanced data structures are usually records that hold values and pointers/references to other (or even the same) kinds of data structures. The basic control flow primitives for traversing these structures are various loops (`while`, `for`, `do` ... `while`) and branch/jump statements (`if`, `switch`, `goto`). The most important operation is assignment, which changes the value of a variable (and thus variables do actually vary, like the name suggests [6]). Computation is modeled by a series of operations which change the global state.

In functional languages the basic data structures are created using the mechanism of algebraic data types – elements of each type so defined are trees whose node labels, branching and types of values held are specified by the user. The basic control flow primitives are pattern matching (checking the label and values of the tree's root) and recursion. The most important operation is function composition, which allows building complex functions from simpler ones. Computation is modeled by substitution of arguments for the formal parameters of a function. Variables don't actually vary – they are just names for expressions. [6]

```
int sum(int[] a)
{
    int result = 0;
    for(int i = 0; i < a.length; ++i)
    {
        result += a[i];
    }
    return result;
}
```

Listing 1: A simple program for summing all integers stored in an array, written in an imperative pseudocode that resembles Java.

```
data List a = Nil | Cons a (List a)

sum : List Int -> Int
sum Nil = 0
sum (Cons x xs) = x + sum xs
```

Listing 2: A simple program for summing all integers stored in a (singly-linked) list, written in a functional pseudocode that resembles Haskell.

The two above programs showcase the relevant differences in practice. In both cases we want a function that sums integers stored in the most basic data structure. In the case of our pseudo-Java, this is a built-in array, whereas in our pseudo-Haskell, this is a list defined using the mechanism of algebraic data types, as something which is either empty (`Nil`) or something that has a head of type `a` and a tail, which is another list of values of type `a`.

In the imperative program, we declare a variable `result` to hold the current value of the sum and then we loop over the array. We start by creating an iterator variable `i` and setting it to `0`. We successively increment it with each iteration of the loop until it gets bigger than the length of the array and the loop finishes. At each iteration, we modify `result` by adding to it the array entry we're currently looking at.

In the functional program, we pattern match on the argument of the function. In case it is `Nil` (an empty list), we declare the result to be `0`. In case it is `Cons x xs` (a list with head `x` and tail `xs`), we declare that the result is computed by adding `x` and the recursively computed sum of numbers from the list `xs`.

Even though these programs are very simple, they depict the basic differences between imperative and functional algorithms quite well. Some less obvious differences are as follows.

First, functional data structures are by default immutable and thus persistent [7], whereas this is not the case for imperative data structures – they have to be explicitly designed to support persistence. This means implementing some techniques, like backtracking, is very easy in functional languages, but it often requires much more effort in imperative languages. The price of persistence often is, however, increased memory usage.

Second, pointer juggling in imperative languages allows a more efficient implementation of some operations on tree-like structures than using algebraic data types, because nodes in such trees can have pointers not only to their children, but also to parents, siblings, etc. The most famous data structure whose functional, asymptotically optimal implementation is not known is union-find. [8]

The third point is that arrays, the bread-and-butter of imperative programming, provide random access read and write in $O(1)$ time, whereas equivalent functional random access structures work in $O(\log n)$ time where n is the array size (or, at best, $O(\log i)$ where i is the accessed index). This means that algorithms relying on constant time random access will suffer an asymptotic performance penalty when implemented in functional languages. [9]

Even though this asymptotic penalty sounds gloomy, not all hope is lost, because of two reasons. First, mutable arrays can still be used in purely[3] functional languages if the mutability is hidden behind a pure interface. An example of this is Haskell's ST monad. [10] Second, functional languages that are impure, like Standard ML or OCaml, allow using mutable arrays without much hassle, which often saves the day.

1.3 The many-worlds interpretation of imperative algorithms

We stated earlier that we intend to concentrate on formally proving correctness of purely functional algorithms. Before doing that, we take a short detour to look at how it differs from the activities and workflows of the usual algorithmic business,[4] what we embrace and what we're leaving out.

When an algorithmist encounters a problem, let's say "How do I sort a list of integers?", he will follow a path towards the solution which looks roughly like this:

- Formulate a (more) precise specification of the problem.

---

[3] "Pure" and "impure" are loose terms used to classify functional languages. "Pure" roughly means that a language makes organized effort to separate programs that do ordinary number crunching or data processing from those that have side effects, like throwing exceptions or connecting to a database. An "impure" languages doesn't attempt at such a separation.

[4] A person engaging in the usual algorithmic business we will call an *algorithmist*.

- Design an algorithm that solves the problem and write it down using some kind of pseudocode, keeping a good balance between generality and detail.
- Prove that the algorithm is correct. If a proof is hard to find, this may be a sign that the algorithm is incorrect.
- Analyze complexity of the algorithm, i.e. how much resources (number of steps, bits memory, bits of randomness, etc.) does the algorithm need to solve the problem of a given size. If the algorithm needs too much resources, go back and try to design another one that needs less.
- Implement the algorithm and test whether the implementation is correct.
- Run some experiments to assess the actual performance of the implementation, preferably considering a few common scenarios: random data, data that often occurs in the real world, data constructed by an evil adversary, etc. If the performance is unsatisfying, try to find a better implementation or go back and design a better algorithm.

Of course this recipe is not stiff and some variations are possible. The two most popular ones would be:

- The extremely practical, in which the specification and proof are dropped in favour of the test suite, the pseudocode is dropped in favour of the actual implementation, and the complexity analysis is only cursory and performed on the go during the design phase, in the algorithmist's head. This approach permeates comeptitive programming, because of the time pressure, and industry, because most "algorithms" there amount to unsophisticated data processing that doesn't need specification or proof.
- The extremely theoretical, in which there is no implementation and thus no tests and no performance assessment, and the most time-consuming part becomes the complexity analysis. This approach is widespread in teaching, where the implementation part is left to students as an exercise, and in theoretical research, where real-world applicability is not always the most important goal.

No matter the exact recipe, there is a very enlightening thing to be noticed, namely all the different worlds in which these activites take place. For example, the algorithm design process takes place in the algorithmist's mind, but after he's done, it is usually written down in some kind of comfortable pseudocode that allows skipping inessential detail. The actual implementation, in contrast, will be in some concrete programming language – a very popular one among algorithmists is `C++`.

The specification and the proof are usually written in English (or whatever the language of the algorithmist), intermingled with some mathematical symbols for numbers, relations, sets and quantifiers, but that's only the surface view. If we take a closer look, it will most likely turn out that the mathematical part is based on classical first-order logic[5] and some kind of naive set theory. When pressed a bit, however, the algorithmist will readily assert that the set theory could be axiomatized using, let's say, the Zermelo-Fraenkel axioms.

The next hidden world, in which the complexity analysis takes its place, is the model of computation. In most cases it is not mentioned explicitly, just like logic and set theory in the previous paragraph, but usually it's easy to guess. Most algorithms are, after all, intended to be implemented on modern computers, and the model of computation most similar to the workings of real hardware is the RAM machine.

As we see, the typical solution of an algorithmic problem stems from a sequence (or rather, a loop) of activities which live in six different worlds: the world of abstract ideas, represented by the pseudocode, the world of programming, represented by the implementation language, the world of formal math, the world of informal math, the world of idealized execution, represented by the model of computation, and the world of concrete execution, represented by the hardware.

Even though the above many-worlds recipe and its variations work well in practice, as evidenced by the huge number of algorithms humanity put to use for solving its everyday problems, an inquisitive person interested in formal verification (or perhaps a philosopher) could ask a plethora of questions about how all these worlds fit together:

- Does the implementation correspond to the pseudocode?
- Could the informally stated specification and proof really be cast into the claimed formal system?
- Does the model of computation actually model the hardware well?
- Could the model of computation (and the complexity analysis) be formalized?
- Assuming the implementation language is compiled, how is the source program related to machine code executed by the hardware, i.e. is compilation correct?
- Does the analysis agree with the implementation language semantics?[6]

Each of these questions can be seen as pointing at a link between two worlds and each such link is a potential source of errors – the worlds may not correspond too well. Because each pair of worlds can give rise to a potential mismatch, in theory there is a lot to worry about.

---

[5] By "classical" we mean that the logic admits nonconstructive reasoning principles like proof by contradiction [11] – its use can be seen, for example, in proofs of optimality.

[6] If it's the implementation that is analyzed and not the pseudocode.

We allowed ourselves to wander into this lengthy overview of the usual algorithmic business in order to contrast it with the approach of formally verified functional algorithms, which is an attempt at getting rid of potential world mismatch errors by transfering (nearly) all of the activities into a single, unified, formal world.

1.4 Formal verification as a world-collapser

This formal world is Coq [12]. Coq is a proof assistant – a piece of software whose goal is helping to state and prove mathematical theorems. This help consists of providing a formal language for doing so, called Gallina, a language for automating trivial proof steps and writing proof search procedures, called Ltac, and a languages of commands, called Vernacular, which simplifies tasks like looking up theorems from the standard library. The Coq ecosystem also has many useful tools, like CoqIDE, an IDE with good support for interactive proving.

Thanks to the Curry-Howard correspondence [13], all of these great things can also be interpreted from the programmer's perspective. Gallina is a dependently typed functional programming language, Ltac is a language for automatic program synthesis, and Vernacular offers a suite of facilities similar to those found in most IDEs. CoqIDE can be seen as supporting interactive program construction (and interactively proving a program correct can be seen as a very powerful form of debugging!).

Applied to algorithms, Coq gives us numerous benefits. First, we no longer need to do pen-and-paper specifications and proofs, so the world of informal mathematics gets eliminated from the picture. Second, it has very powerful abstraction capabilities, which bypass the need for pseudocode – we can directly implement the high-level idea behind the algorithm (an example of this will be provided in Chapter 2). This merges the worlds of ideas and programming into one. Third, as already mentioned, the Curry-Howard correspondence unifies functional programming and constructive mathematics, which further collapses the two already-collapsed worlds.

What we are left with are just three worlds instead of the six we had at the beginning: one for programming, proving and expressing high-level algorithmic ideas, one for the analysis (model of computation), and one for execution (hardware). Most of the links between activities now live in the first world and we can use the power of formal proofs to make sure there are no mismatches. We can prove that the algorithm satisfies the specification. If we decide to have more than one implementation (for example when we're looking for the most efficient one), we can prove they are equivalent. We can even prove that the specification is "correct", i.e. that the object it describes is unique. To sum up, we can avoid the various mismatches of previous section using formal proofs inside our unified world.

But as long as not all worlds were collapsed into one, there is still some room for error. First, becuase Coq is implemented in OCaml, there is no guarantee that our formal proofs are absolutely correct. In fact, as can be seen from `https://github.com/coq/coq/blob/master/dev/doc/critical-bugs`, Coq has experienced about one critical bug per year. This means if we want absolute certainty in our proofs, we need to verify Coq's implementation by hand, so the world of informal math is still somewhat alive.

The second point, related to the first, is that there is no guarantee the semantics of code executed by the hardware is the same as the semantics of the source code. There have been some attempts at formalizing various aspects of Coq in Coq [14] [15] to this effect, but they have the status of exploratory research or work-in-progress. A more serious barrier that prevents us from ever fully formalizing Coq in Coq is Gödel's incompleteness theorem.

Third, the model of computation still lives in a separate world, formally related neither to the code nor to the hardware. What's worse, there is a huge chasm between the preferred models of computation used in computational complexity (Turing machines [16]) and algorithms (RAM machines) on the one hand, and the theory of programming languages (lambda calculi [17]) on the other hand – see [18] for a brief overview.

Regarding the third point, there has been some ongoing research whose goal is to bring lambda calculus on par with other models of computation, with interesting results like [19] showing that it's not as bad as complexity theorists think, but it's still very far from entering the mainstream consciousness. Another related line of research is implicit complexity theory – an attempt to characterize complexity classes not in terms of resources needed by a machine to compute a function, but in terms of restricted, minimalistic programming languages that can be used to implement functions from these classes. See [20] [21] for an introduction. A third related body of research concerns cost semantics, a technique for specifying the resources needed by a programming language's constructs together with their operational behaviour, so that one no longer needs to consider the model of computation (or rather, the programming language becomes the model of computation). This opens up some interesting directions, like automatic complexity analyses during compilation time. See [22] for arguments why this is a viable alternative to machine-based computation models and [23] for an example paper using this technique.

We will not worry about the concerns raised, because a single formal world is still a huge win in terms of correctness. Regarding point 1, in practice, if Coq accepts a proof, then the proof is correct – chances of running into a critical bug by accident are minimal. We're also not as much interested in running algorithms and analyzing their complexity, so points 2 and 3 don't matter to us at all.[7]

---

[7] This of course does not mean that we will study silly, exponentially slow algorithms...

1.5 Type theory (as a model of computation)

In this section we describe the basic principles on which Coq is founded. Coq is based on a formal system called Calculus of Constructions (CoC)[24], which later evolved into the Calculus of Inductive Constructions (CIC) [25] and finally into Predicative Calculus of Cumulative Inductive Constructions (pCuIC) [26].

To avoid drowning in minute details, we instead describe Martin-Löf Type Theory (MLTT) [27] [28], a closely related system which from now on we will call simply "type theory". In the next section, where we introduce Coq, we will point out the relevant differences from type theory as we go. The material presented in this section is standard, but in the light of previous section's closing paragraphs we encourage people familiar with type theory to read it as a description not of a formal system, but of a model of computation.

In type theory, as the name suggests, the basic objects of interest are types. In order to be meaningful, a type must first be formed. For example, we can always form the type $\mathbb{N}$ of natural numbers, but in general this is not the case: sometimes to form a type we must make an assumption or even several assumptions. We keep track of our assumptions using contexts. For example, if in a context Γ we can form the types A and B, written $\Gamma \vdash A$ and $\Gamma \vdash B$ respectively, then we can form the type of functions from A to B, written $\Gamma \vdash A \to B$. The rules that govern type formation are called, unsurprisingly, formation rules.

After we have formed a type, we can manipulate its terms. For example, the type of natural numbers $\mathbb{N}$ has as terms, among others, 4 and $2 + 2$, written $4 : \mathbb{N}$ and $2 + 2 : \mathbb{N}$, respectively. Similarly to the case of type formation, what is a term and what is not depends on the assumptions we have made: $n + m : \mathbb{N}$ only under the assumptions that $n : \mathbb{N}$ and $m : \mathbb{N}$, which we can formally write as $n : \mathbb{N}, m : \mathbb{N} \vdash n + m : \mathbb{N}$. From now on we won't emphasize context-dependence, as everything we do in type theory depends on context.

Rules for creating terms are divided into two genres: introduction rules and elimination rules. Introduction rules for a type tell us how to create new terms of this type. For example, the introduction rules for $\mathbb{N}$ are $\Gamma \vdash 0 : \mathbb{N}$, which says that 0 is a natural number, and $\Gamma, n : \mathbb{N} \vdash \texttt{succ}(n) : \mathbb{N}$, which says that the successor of n is a natural number given that n is a natural number.

Elimination rules for a type, on the other hand, tell us how to use terms of this type to get terms of other types. For example, the simplest elimination rule for $\mathbb{N}$ says that if $\Gamma \vdash z : X$ and $\Gamma \vdash s : \mathbb{N} \to X$, then $\Gamma, n : \mathbb{N} \vdash \texttt{rec}_{\mathbb{N}}(z, s, n) : X$. This rule, alternatively called a recursor, allows us to define functions by recursion on natural numbers, but to do proofs by induction we need a stronger elimination rule. We won't state it here to keep things simple.

The next thing we are interested in is convertibility, which formalizes the notion of computation. Intuitively, two terms are convertible if they evaluate to the same result. For example, we would expect that both $2 + 2$ and 4 evaluate to 4, and this is indeed the case, so they are convertible. To state that formally, we write $\Gamma \vdash 2 + 2 \equiv 4 : \mathbb{N}$. Rules that govern the behaviour of convertibility, similarly to rules for term manipulation, come in two genres: computation rules and uniqueness rules.

Computation rules describe what happens when we first apply an introduction rule and then an elimination rule. For example, the first computation rule for $\mathbb{N}$ states that given $\Gamma \vdash z : X$ and $\Gamma \vdash s : \mathbb{N} \to X$, we have $\Gamma \vdash \texttt{rec}_{\mathbb{N}}(z, s, 0) \equiv z : \mathbb{N}$, and the second rule states that given $\Gamma \vdash z : X$, $\Gamma \vdash s : \mathbb{N} \to X$ and $\Gamma \vdash n : \mathbb{N}$ we have $\Gamma \vdash \texttt{rec}_{\mathbb{N}}(z, s, \texttt{succ}(n)) \equiv s(\texttt{rec}_{\mathbb{N}}(z, s, n)) : \mathbb{N}$.

Uniqueness rules, on the other hand, describe what happens when we first use an elimination rule and then an introduction rule. For example, the uniqueness rule for $\mathbb{N}$ states that given $\Gamma \vdash n : \mathbb{N}$, we have $\Gamma \vdash \texttt{rec}_{\mathbb{N}}(0, \texttt{succ}, n) \equiv n : \mathbb{N}$. Note, however, that this uniqueness rule for natural numbers is rarely included in most presentations of type theory.

The last thing to mention is how exactly contexts work and some of its consequences. First, there is the empty context, usually denoted by leaving empty space to the left of $\vdash$, for example $\vdash \mathbb{N}$ means that we can form the type $\mathbb{N}$ in the empty context. Second, if we have a context Γ and in this context we can derive $\Gamma \vdash A$, then we can extend the context Γ with a variable of this type, written $\Gamma, x : A$, provided that the name x doesn't already occur in Γ.

The most important consequence of this is that we need a notion of convertibility for types. Because types are formed in contexts, and contexts can contain assumptions of the form $x : A$, types can depend on terms. For example, for any natural number n we can form the type $\texttt{Fin}(n)$ that has exactly n elements, formally written $\Gamma, n : \mathbb{N} \vdash \texttt{Fin}(n)$. This raises the question: is the type $\texttt{Fin}(2 + 2)$ the same as the type $\texttt{Fin}(4)$? The answer is yes – because $2 + 2$ and 4 are converible, these types are also convertible, formally written $\Gamma \vdash \texttt{Fin}(2 + 2) \equiv \texttt{Fin}(4)$. The rules that govern type convertibility aren't very interesting – they say that convertibility is an equivalence relation and that dependent types are convertible whenever the terms they depend on are convertible.

An astute reader has probably noticed that we sneaked into the above description a word that is familiar but was not defined, namely “element”. The elements of a type A are its closed terms that are in normal form. A term is closed if its context is empty and it is in normal form if it is the final result of evaluation,[8] i.e. it can't be evaluated any further.

For example, $\vdash 4 : \mathbb{N}$ is a closed term in normal form, $\vdash 2 + 2 : \mathbb{N}$ is a closed term that is not in normal form and $n : \mathbb{N} \vdash \texttt{succ}(n) : \mathbb{N}$ is a term in normal form, but it is not closed because it depends on the assumption $n : \mathbb{N}$ (such terms are called open). It turns out that for natural numbers, the elements are precisely $0, \texttt{succ}(0), \texttt{succ}(\texttt{succ}(0)), \texttt{succ}(\texttt{succ}(\texttt{succ}(0))), \ldots$ which we can interpret as $0, 1, 2, 3, \ldots$ respectively. So, the elements of the type of natural numbers are precisely the natural numbers. Who would have guessed!

---

[8] We leave the idea of evaluation informal and won't describe it in more detail.

That's it. The basic idea behind type theory is very simple. Of course the above presentation is far from being complete – to be, we would have to list all the formation, introduction, elimination, computation and uniqueness rules for all types, and also dozens of boring technical rules whose role is to make sure everything works as expected.

The explanation of elements prompts us to change our perspective on type theory from a model of computation back to foundational and pose the following question: what is the relation between type theory and set theory? After all, types have elements and sets have elements too. Below, we provide a short comparison.

Set theory has two basic kinds of objects – propositions and sets – living in two separate layers. Propositions live in the logical layer, which consists of first order classical logic, whereas sets live in the set-theoretical layer, which consists of set theory axioms. Type theory, on the other hand, has only one basic kind of objects, namely types, and only one layer – the type layer. Both propositions and sets can in type theory be interpreted using types.

In set theory, the membership predicate $x \in A$ is a proposition which can be proved or disproved. It is decidable internally, because of the law of excluded middle, but not externally, because there is no computer program that can tell us whether $x \in A$. In type theory, $x : A$ is not a proposition, but a judgement, which means one can establish that it holds, but it makes no sense to negate it. It neither makes sense to talk about it's internal decidability. However, it is externally decidable, which means there is a computer program that checks whether it holds.

In set theory, sets are semantic entities that need not, in general, be disjoint, so an element can belong to many sets. In type theory, types are syntactic entities that are always disjoint. Elements can belong to only one type, or, to be more precise, if an element belongs to two types, then these types are convertible.

1.6 Way of the Coq

In this section we give a very short Coq tutorial. Before reading it, the unfamiliar reader should install CoqIDE[9] and run the listing below[10] in interactive mode. This experience will greatly enrich the explanation.

---

[9] Available from [12].

[10] It can be found in the thesis' repository `https://github.com/wkolowski/RandomCoqCode`in the directory `Thesis/Snippets/`

```
Require Import List Lia.
Import ListNotations.

Print list.
(*
Inductive list (A : Type) : Type :=
    | nil : list A
    | cons : A -> list A -> list A
*)

Print app.
(*
fix app {A : Type} (l1 l2 : list A) {struct l1} : list A :=
match l1 with
  | [] => l2
  | h :: t => h :: app t l2
end
*)

Record rev_spec {A : Type} (f : list A -> list A) : Prop :=
{
    f_app   : forall l1 l2 : list A, f (l1 ++ l2) = f l2 ++ f l1;
    f_singl : forall x : A, f [x] = [x];
}.

Theorem rev_spec_unique :
  forall {A : Type} (f g : list A -> list A),
    rev_spec f -> rev_spec g ->
      forall l : list A, f l = g l.
Proof.
  intros A f g [Hfc Hfs] [Hgc Hgs].
  induction l as [| h t]; cbn.
    specialize (Hfc [] []); specialize (Hgc [] []).
      apply (f_equal (@length _)) in Hfc;
      apply (f_equal (@length _)) in Hgc.
      cbn in *. rewrite app_length in *.
      destruct (f []), (g []).
        reflexivity.
        1-3: cbn in *; lia.
    change (h :: t) with ([h] ++ t).
      rewrite Hfc, Hgc, Hfs, Hgs, IHt. reflexivity.
Qed.
```

We start by importing modules for working with lists – we will need some list functions and notations (Coq allows defining custom notations using a built-in notation mechanism), and also `lia`, a procedure for reasoning in **l**inear **i**nteger **a**rithmetic (hence the name).

The command `Print` is part of the Vernacular – a language of useful commands for looking up definitions and theorems, and other things usually provided at the IDE level. We use it to recall the definition of `list` and `app` (we modified the outputs of these commands to simplify our explanations).

`list` is a parameterized family of inductive types. This means that for each type `A`, there is a type list `list A`, defined by giving the possible ways to construct its elements. An element of this type can be either `nil`, which represents an empty list, or `cons h t` for some `h` : `A` and `t` : `list A`, which represents a list with head `h` and tail `t`. This is very similar to the definition of lists in pseudo-Haskell that we saw in section 2.

`app` is a function that concatenates two lists. It takes as arguments a type `A` and two lists of elements of type `A`, named `l1` and `l2`, and it returns an element of type `list A` as result. The first argument is surrounded by curly braces, which means it is implicit – we won't need to write it, because Coq can infer it on its own. The other arguments are surrounded by parentheses, which means they are explicit – we need to provide them when applying the function.

`app` is defined by recursion, as indicated by the keyword `fix`, and its termination is guaranteed because this recursion is structural, as indicated by the annotation `struct l1`. The definition goes by pattern matching on the argument `l1`. if it is a `nil`, the result is `l2` and if it is `h` :: `t`, then the result is `h` :: `app t l2`. Here the double colon `::` is a handy notation for the constructor `cons`.

In section 3 we said that in a formal setting we can prove a specification correct. This is actually the first of our techniques and in the rest of the listing our task will be to demonstrate how it works. The thing we want to specify is a function for reversing lists and the specification, named `rev_spec`, starts on line 20. It takes as arguments a type `A` and a function `f` : `list A -> list A` and our intention is that `rev_spec f` means "`f` is a function that reverses lists".

The definition says that `rev_spec f` is a proposition[11] and that it is defined to be a record consisting of two fields. The first one is named `f_app` and intuitively means that `f` anticommutes with list concatenation – if we concatenate two lists and then reverse the result using `f`, it's the same as if we first reversed the lists and then concatenated them in the opposite order. The second field is named `f_singl` and means that on lists with only one element `f` acts like an identity function.

---

[11] This is actually one of the differences between type theory as presented in the last section and Coq. In type theory, we represent propositions using ordinary types. In Coq, only types of sort `Prop` are propositions, whereas types of sort `Type` are not.

Now, let's take a break from Coq and make a short conceptual trip: how can we prove that a specification specifies precisely what we wanted? Of course, in general, we can't, because what we really want lives in the world of ideas, whereas the specification lives in the formal world. But if we are sure that the desired solution will meet the specification, the best sanity check we can perform is proving that the specification determines a unique object. In such a situation if we are dissatisfied with the solution, we are sure that it is not because of some bug or other kind of technical error, but simply because we were unable to express what we wanted.

Back to Coq, the theorem `rev_spec_unique` is the fulfillment of our plan of "proving the specification correct". Such a theorem in Coq has the same status as an ordinary definition – the statement of the theorem is a type and to prove it we need to construct an element of that type. However, the means of construction are different from those typically used for definitions: we don't provide the proof term explicitly, like the body of `app` between `match` and `end` was provided explicitly, but instead we use Coq's tactic language called Ltac.

In Ltac, we can manipulate tactics, which can be seen as things that encapsulate a particular line of mathematical reasoning and argumentation (or, from the programmer's perspective, as techniques for synthesizing particular kinds of programs). The most basic tactics directly correspond to introduction and elimination rules of type theory from the previous section and to basic features of Coq's term language, like pattern matching. More complex tactics can be created from simpler ones by using features provided by Ltac, like various kinds of tactic combinators, searching hypotheses and variables in the proof context, various flavours of backtracking and the ability to inspect Coq's terms as syntax. The last of these is something that is not possible using the term language, as it would lead to contradiction.

1.7 An ultra short literature review

Chapter 2

A quick *tour de* sort

Chapter 3

A man, a plan, a canal – MSc thesis

3.1 Design

- First step: specification (describe the path from intuition to formal spec).
- Second step: design (describe the path from the concrete to the abstract by tracing a stub proof of correctness).
- We shouldn't require proofs in order to run programs. Clou: packed vs unpacked classes/records/modules.

3.2 Techniques

- General recursion: Bove-Capretta method as the way to go.
- Functional induction as the way-to-go proof technique. Mention the Equations plugin.

3.3 Topics

- Quicksort: in functional languages we have so powerful abstractions that we can actually implement algorithms and not just programs.
- Braun mergesort: in order not to waste resources, we sometimes have to reify abstract patterns, like the splitting in mergesort.
- Binary heaps: a case study to show the basic workflow and that it's not that obvious how to get basic stuff right.
- Cool data structures: ternary search trees, finger trees.

Chapter 4

Conclusion

Mention thesis' repository: `wkolowski.github.io/RandomCoqCode/Thesis/`

Bibliography

[1] `https://www.thefreedictionary.com/algorithm`

[2] *Do "algorithms" exist in Functional Programming?*, `https://stackoverflow.com/questions/25940327/do-algorithms-exist-in-functional-programming`

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Introduction to Algorithms*, `http://ressources.unisciel.fr/algoprog/s00aaroot/aa00module1/res/%5BCormen-AL2011%5DIntroduction_To_Algorithms-A3.pdf`

[4] Donald Knuth, *The Art of Computer Programming*

[5] Thomas S. Kuhn, *The Structure of Scientific Revolutions*

[6] Robert Harper, *Words Matter*, 2012 `https://existentialtype.wordpress.com/2012/02/01/words-matter/`

[7] Driscoll JR, Sarnak N, Sleator DD, Tarjan RE *Making data structures persistent*, 1986

[8] Sylvain Conchon, Jean-Christophe Fillâtre, *A Persistent Union-Find Data Structure*, 2007 `https://www.lri.fr/~filliatr/ftp/publis/puf-wml07.pdf`

[9] Nicholas Pippenger, *Pure versus impure Lisp*, 1996 `https://www.cs.princeton.edu/courses/archive/fall03/cs528/handouts/Pure%20Versus%20Impure%20LISP.pdf`

[10] John Launchbury, Simon Peyton Jones, *Lazy Functional State Threads*, 1994 `https://www.microsoft.com/en-us/research/wp-content/uploads/1994/06/lazy-functional-state-threads.pdf`

[11] Andrej Bauer, *Proof of negation and proof by contradiction*, 2010 `http://math.andrej.com/2010/03/29/proof-of-negation-and-proof-by-contradiction/`

[12] `https://coq.inria.fr/`

[13] Morten Heine Sørensen, Paweł Urzyczyn,
Lectures on the Curry-Howard Isomorphism, 2006,
`http://disi.unitn.it/~bernardi/RSISE11/Papers/curry-howard.pdf`

[14] Bruno Barras, Benjamin Werner, *Coq in Coq*, 1997,
`http://www.lix.polytechnique.fr/Labo/Bruno.Barras/publi/coqincoq.pdf`

[15] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, Théo Winterhalter,
Coq Coq Correct! Verification of Type Checking and Erasure for Coq, in Coq, 2020,
`https://www.irif.fr/~sozeau/research/publications/drafts/Coq_Coq_Correct.pdf`

[16] Alan Turing, *On Computable Numbers, with an Application to the Entscheidungsproblem*, 1936
`https://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf`

[17] Alonzo Church, *An Unsolvable Problem of Elementary Number Theory*, 1936
`https://www.ics.uci.edu/~lopes/teaching/inf212W12/readings/church.pdf`

[18] Guy Blelloch, Robert Harper,
λ-Calculus: The Other Turing Machine, 2015
`https://www.cs.cmu.edu/~rwh/papers/lctotm/cs50.pdf`

[19] Ugo Dal Lago, Simone Martini,
The Weak Lambda Calculus as a Reasonable Machine, 2008
`https://www.di.unito.it/~deligu/CDR60_TCS/Martini.pdf`

[20] Ugo Dal Lago,
A Short Introduction to Implicit Computational Complexity, 2010
`http://cs.unibo.it/~dallago/FICQRA/esslli.pdf`

[21] Ugo Dal Lago, *Machine-Free Complexity*, 2019
`https://caleidoscope.sciencesconf.org/data/DalLago_caleidoscopeslides.pdf`

[22] Robert Harper, *Languages and Machines*, 2015
`https://existentialtype.wordpress.com/2011/03/16/languages-and-machines/`

[23] Norman Danner, Daniel R. Licata, Ramyaa Ramyaa
Denotational Cost Semantics for Functional Languages with Inductive Types
`https://dlicata.wescreates.wesleyan.edu/pubs/dlr15inductive/dlr15inductive.pdf`

[24] Thierry Coquand, Gérard Huet,
The calculus of constructions, 1984,
`https://www.sciencedirect.com/science/article/pii/0890540188900053`

[25] Christine Paulin-Mohring,
Introduction to the Calculus of Inductive Constructions, 2015
`https://hal.inria.fr/hal-01094195/document`

[26] Amin Timany, Matthieu Sozeau,
Consistency of the Predicative Calculus of Cumulative Inductive Constructions (pCuIC), 2018
`https://hal.inria.fr/hal-01615123v2/document`

[27] Per Martin-Löf,
An intuitionistic theory of types, 1972
`https://archive-pml.github.io/martin-lof/pdfs/An-Intuitionistic-Theory-of-Types-1972.pdf`

[28] Per Martin-Löf,
Intuitionistic Type Theory, 1984
`https://archive-pml.github.io/martin-lof/pdfs/Bibliopolis-Book-retypeset-1984.pdf`

[29] Benjamin C. Pierce, Andrew W. Appel et al.,
Software Foundations, 2019,
`https://softwarefoundations.cis.upenn.edu/`

[30] Yves Bertot and Pierre Castéran,
Interactive Theorem Proving and Program Development
Coq'Art: The Calculus of Inductive Constructions, 2004,
`https://www.labri.fr/perso/casteran/CoqArt/`

[31] Adam Chlipala, *Certified Programming with Dependent Types*,
`http://adam.chlipala.net/cpdt/`