2 Coq and SSReflect

We decided to employ the Small Scale Reflection Extension [18] (**SSReflect**[1]) for the **Coq**[2] proof assistant [26]. The most important factors in this decision were SSReflect's excellent support for finite types, list operations and graphs. SSReflect also introduces an alternative scripting language that can often be used to shorten the bookkeeping overhead of proofs considerably.

2.1 Coq

The Coq manual [26] states the following about Coq: The Coq system is designed to develop mathematical proofs, and especially to write formal specifications, programs and to verify that programs are correct with respect to their specification. It provides a specification language named Gallina. Terms of Gallina can represent programs as well as properties of these programs and proofs of these properties. Using the so-called Curry-Howard isomorphism, programs, properties and proofs are formalized in the same language, which is a λ-calculus with a rich type system called Calculus of Inductive Constructions. All logical judgments in Coq are typing judgments. The very heart of the Coq system is the type-checking algorithm that checks the correctness of proofs, i.e that a program complies to its specification. Coq also provides an interactive proof assistant to build proofs using specific programs called tactics.

2.2 SSReflect

SSReflect is a set of extensions to the proof scripting language of the Coq proof assistant. They were originally developed to support small-scale reflection. However, most of them are of quite general nature and improve the functionality of Coq in most basic areas such as script layout and structuring, proof context management and rewriting [18].

SSReflect comes with an extensive library [17] covering many mathematical concepts leading up to finite group theory. In fact, we barely scratch the surface of the library in this development. The interested reader may convince herself/himself of the sheer size of the library[3].

[1] `http://www.msr-inria.inria.fr/Projects/math-components`
[2] `http://coq.inria.fr/`
[3] `http://coqfinitgroup.gforge.inria.fr/`

2.2.1 Boolean Reflection

SSReflect offers boolean reflections for decidable propositions, so called "small-scale reflection". A term of type `reflect P b` is a proof of the equivalence of the boolean statement `b` and the proposition `P`. SSReflect has built-in support to change from boolean to propositional statements if they are equivalent. This allows us to always assume the most convenient perspective in our proofs.

2.2.2 Boolean Predicates

SSReflect has special type for boolean predicates, `pred T := T -> bool`, where `T` is a type. We make use of SSReflect's syntax to specify boolean predicates. This allows us to specify predicates in a way that resembles set-theoretic notation, e.g. `[pred x | <boolean expression in x> ]`. Furthermore, we can use the functions `pred1` and `pred0` to specify the singleton predicate and the empty predicate, respectively. The complement of a predicate can be written as `[predC p]`. The syntax for combining predicates is `[pred? p1 & p2 ]`, with `?` being one of `U` (union), `I` (intersection) or `D` (difference). For predicates given in such a way, we write `y \in p` to express that `y` fulfills `p`. There is also syntax for the preimage of a predicate under a function which can be written as `[preim f of p]`.

There are also applicative (functional) versions of of `predC`, `predU`, `predI`, `predD`, which are functions that take predicates as arguments and return predicates. `pred1 x` represents the predicate `[pred y | y == x]`.

2.2.3 Finite Types

The most important feature of SSReflect for our purpose are finite types. Finite types are types that have a finite number of elements. The type of finite types is `finType`. SSReflect offers a number of important operations on finite types. Finite types are closed under basic operations such as `option` and `sum`. SSReflect provides boolean versions of the universal and existential quantifiers on finite types, `[ forall x, p x]` and `[exists x, p x]`, where `p` is a boolean predicate. We compute the number of elements for which a predicate `p` on a finite type returns true with `#|p|`. `enum p` gives a list of these elements. A finite type in the position of a predicate coerces to a predicate that is always true.

We can also create finite types from lists. Instances of these types can be specified with the `SeqSub` constructor, which takes as arguments an element of the list and a proof that this element is contained in the list.

2.2.4 Finite Sets

SSReflect also supports finite sets, which are sets over finite types. Finite sets themselves are finite types, which allows us to use them in the construction of finite automata. Additionally, they come with the same syntax for counting and enumerating their elements that we introduced for finite types. The type of a finite set is `{set T}`, where `T` is

a finite type. The function set1 constructs a singleton set containing its first argument. The type of the resulting finite set can be inferred from the argument. The union over finite sets can be expressed by \bigcup_(x | P x) F, which corresponds to the mathematical notation $\bigcup_{x \in X, Px} F$, with X being the (finite) type of x. More information about these so-called "big operators" can be found in [10].

2.2.5 Equality

We can use f =1 g to express that the functions f and g agree in all arguments. If we regard f and g as sets, we can write f =i g, which is defined as forall x, x \in f = x \in g. We need to do this because Coq's equality = is intensional, which means that even if we have f =1 g, we are not, in general, able to prove f = g. Thus, we use =1 or =i in Coq, when we write = mathematically. This expresses the notion of extensional equality of classical mathematics.