Chapter 2

Coq and SSReflect

We decided to employ the Small Scale Reflection Extension (**SSReflect**[1]) for the **Coq**[2] proof assistant. 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

Description, citation

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 [16].

2.2.1 Finite Types

The most important feature of SSReflect for our purpose are finite types. Finite types are types that have a finite number of inhabitants. Their implementation is based on lists. Every element of a finite type is contained in the associated (de-duplicated) list and vice versa. SSReflect's support for finite types is based on canonical structures, instances of which come predefined for basic finite types and type constructors. This allows us to easily combine basic finite types such as `bool` with type constructors such as `option` and `sum`.

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

SSReflect provides boolean versions of the universal and existential quantifiers on finite types, **forallb** and **existsb**. We can compute the number of elements in a finite type F with #|F|. enum gives a list of all items of a finite type.

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

2.2.2 Boolean Reflection

SSReflect offers boolean reflections for decidable propositions. This allows us to switch back and forth between equivalent boolean and propositional predicates.

2.2.3 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.

2.2.4 Miscellaneous

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. Coq's equality = is intensional, which means that even if we have f =1 g, we will not, in general, be able to proof f = g. Thus, we will use =1 and =i in Coq, when we write = mathematically.