A Comparative Study of Coq and HOL

Vincent Zammit

Computing Laboratory, University of Kent, Canterbury, Kent, UK

Abstract. This paper illustrates the differences between the style of theory mechanisation of Coq and of HOL. This comparative study is based on the mechanisation of fragments of the theory of computation in these systems. Examples from these implementations are given to support some of the arguments discussed in this paper. The mechanisms for specifying definitions and for theorem proving are discussed separately, building in parallel two pictures of the different approaches of mechanisation given by these systems.

1 Introduction

This paper compares the different theorem proving approaches of the HOL [10] and Coq [5] proof assistants. This comparison is based on a case study involving the mechanisation of parts of the theory of computation in the two systems. This paper does not illustrate these mechanisations but rather discusses the differences between the two systems and backs up certain points by examples taken from the case studies.

One motivation of this work is that many users of theorem provers lack the perspective of knowing more than one such system, mainly due to the amount of time needed to master any such system. Having a single text which builds up pictures of two different systems in parallel allows users of one system to grasp better how the different approach of the other system affects the way theories are mechanised. As a result, knowing the main differences beforehand facilitates the process of learning the other system, and gives a better perspective of the system the user is familiar with.

The case studies are illustrated separately in [26] and in [27]. The mechanisation in HOL is based on the Unlimited Register Machine (URM) model of computation [7], and the main result of the formalisation is a proof that partial recursive functions are URM computable. The mechanisation in Coq is based on a model of computation similar to the partial recursive function model and includes a constructive proof of the S_n^m theorem. Both implementations are in the order of 10,000 lines of code. HOL90 version 7 and Coq version 5.10 were used for the mechanisations.

The two systems are introduced in the next section where a brief overview of each of them is given. Since we are considering the differences between how actual mechanisations of theories are performed in practice, this comparative study treats the mechanisms for definitions (section 3) and theorem proving (section 4) separately. Other considerations are then discussed in section 5 and the last section gives some concluding remarks.

It is noted in this comparative study that the strongest point of the Coq system is the power of the logic it is based on. The HOL logic is much more simple but users can rely on a greater flexibility offered by the metalanguage. As a result HOL theorem proving is much more implementation oriented, while in Coq unnecessary implementation is avoided and discouraged by having a specification and proof language which bridges the user from the metalanguage. These points are built gradually in the following sections and are discussed in the conclusion.

2 An Overview of Coq and HOL

Both systems are based on the LCF [9] style of theorem proving, where all logical inferences are performed by a simple core engine. A metalanguage is provided so that users can extend the system by implementing program modules applying the operations of the core engine. The systems differ from each other however by implementing quite different logics and through the flexibility by which users are allowed to extend the system.

2.1 Coq

The Coq system is an implementation in CAML of the Calculus of Inductive Constructions (CIC) [4], a variant of type theory related to Martin-Löf's Intuitionistic Type Theory [14, 18] and Girard's polymorphic λ-calculus F_ω [8]. Terms in CIC are typed and types are also terms. Such a type theory can be treated as a logic through the *Curry-Howard isomorphism* (see [25, 18] for introductions of the Curry-Howard isomorphism) where propositions are expressed as types. For instance, a conjunction $A \wedge B$ is represented by a product type $A \times B$, and an implication $A \Rightarrow B$ is represented by a function type $A \rightarrow B$. Also, a term of type τ can be seen as a proof of the proposition represented by τ, and thus theorems in the logic are nonempty types. For example, the function

$$curry = \lambda f.\lambda x.\lambda y.f(x,y)$$

which has type $((A \times B) \rightarrow C) \rightarrow A \rightarrow B \rightarrow C$ is a proof of the theorem $((A \wedge B) \Rightarrow C) \Rightarrow (A \Rightarrow B \Rightarrow C)$. Objects which have the same normal form according to $\beta\delta\iota$-conversion are called convertible, and are treated as the same term by the logic. δ-conversion involves the substitution of a constant by its defining term and ι-conversion is automation of inductive definitions. The CIC implemented in Coq differs from that of LEGO [22] by having two sorts of universes, an impredicative universe for sets in which functions are computable, and a predicative universe for types and propositions in which functions (predicates) need not be computable (decidable).

Due to the Curry-Howard isomorphism, theorem proving corresponds to the construction of well typed terms and the core inference engine of Coq is basically a type checking algorithm of CIC terms. Terms whose type is a theorem are usually called proof objects and are stored in Coq theories. The Coq system provides the specification and proof language `Gallina` in which users perform the actual interactive theorem proving. `Gallina` constructs include commands for specifying definitions and for tactic based theorem proving (section 4 discusses this in more detail), and Coq users can extend the `Gallina` language by implementing new contructs in CAML. The files which `Gallina` accepts during theorem proving are usually called scripts (or proof scripts).

2.2 HOL

The HOL system implements (in Standard ML of New Jersey for the case of HOL90) a classical higher order logic based on Church's simple theory of types [3] extended with polymorphic types and inference rules for definitions. Thus, HOL terms are typed, where types represent *nonempty sets* and can be either type constants, type variables (which make the type theory polymorphic), function types (which make the logic higher order) or the application of some type operator to a number of types[1]. Terms are either constants, variables, lambda abstractions or applications; sequents consist of a finite set of terms (the assumptions) and one term (the conclusion), and theorems are sequents which are proved by one of a number of primitive inference rules.

Theorems in the HOL system are represented by an abstract datatype (with name `thm`) having as constructors a small number of functions corresponding to the logic's primitive inference rules. The implementation of this datatype is the core inference engine of HOL, and the type checking mechanism of ML ensures that objects of type `thm` are constructed only by using the type's constructors. Theorem proving in HOL involves the implementation of programs in the metalanguage which yield terms of type `thm`. All support for specifying definitions, constructing types and terms (which can be done by *quotation* in which a system function parses expressions written in a readable syntax into HOL representation), and theorem proving is provided through ML functions which are visible to HOL users. Thus, users can extend the system by implementing new ML functions representing higher level inference rules, decision procedures, proof strategies, definition mechanisms, etc..

3 Definitions

A definition can be considered as a name given to a term or type by which it can be referred to in a theory. For example the definition

$$\texttt{computable}\ n\ f\ =_{def}\ \exists p.\ \texttt{computes}\ p\ n\ f$$

introduces the new object `computable` in the current theory and makes the two expressions $\texttt{computable}\ n\ f$ and $\exists p.\ \texttt{computes}\ p\ n\ f$ for any term f, in some sense interchangeable. In a mechanisation (or formalisation) of a theory, giving definitions is a mechanism by which mathematical concepts are formalised

[1] A function type $\alpha \rightarrow \beta$ can be considered as the application of the operator $\rightarrow$ on the types α and β, and type constants as type operators with arity 0.

by specifying them as being equivalent to expressions containing only already defined terms. The above example illustrates how the concept of a computable n-ary function can be formalised. A concept can also be formalised through the declaration of axioms and both systems allow users to introduce axioms in theories. However, an axiomatic theory can be inconsistent while the definition mechanisms of Coq and HOL guarantee that purely definitional theories are always consistent.

The definition mechanism in Coq introduces new constant names in an environment and allows these terms to be convertible with their defining terms. This applies to both simple abbreviations (δ-conversion) and inductive definitions (ι-conversion). Since proofs and theorems are first class objects in CIC, the name of a theorem is actually a constant definition given to its proof term. In fact, although the specification language `Gallina` gives different constructs for defining terms and for theorem proving, one can, for instance, use tactics to define terms and the definition mechanism to prove theorems. The system differentiates between definitions and theorems by labeling the former objects as *transparent* and the latter as *opaque*. Transparent objects are convertible with their defining terms while opaque objects are not. `Gallina` commands for labeling objects as opaque or transparent are also provided.

The HOL logic treats type and constant definitions differently, and the core system provides one primitive inference rule for type definitions and two for constant definitions. Other inference rules are given for deriving theorems. The function of the HOL primitive rules for definitions is illustrated below, where the differences between the definition mechanism in HOL and in Coq are discussed.

3.1 Type Definitions

The HOL system has one primitive rule for type definitions, which introduces a new type expression α as a nonempty subset of an existing type σ, given a term $P\colon \sigma \to \mathit{bool}$ which denotes its characteristic predicate. However, in practice, the user introduces new types through the type definition package [15] which specifies ML style polymorphic recursive types as well as automatically deriving a number of theorems specifying certain properties about the type (such as the fact that the type constructors are injective).

Such types are specified in Coq by inductively defined sets and types, and the corresponding theorems derived by HOL's type definition package are either returned as theorems by the definition mechanism of `Gallina` or follow from the elimination and introduction rules of the set or type.

The obvious advantage of having types as terms in CIC over HOL's simple type theory is a much more expressive type system which allows quantification over types and dependent types. A dependent type is a type which depends on the value of some particular term. The 'classical' examples of dependent types include $Nat(n)$, the type of the natural numbers less than n, and $\mathit{vector}(A, n)$, the type of vectors (or lists) having n elements of type A. This type is defined inductively in [27]:

```
vector A =_def Vnil: (vector A 0)
           |  Vcons: (n: nat) → A → (vector A n)
                            → (vector A (S n)).
```

and by defining the type for relations

```
Rel =_def λA,B:Set. A → B → Prop.
```

the type of n-ary partial functions over the natural numbers can be defined to be single valued relations between `vector nat` n and `nat`:

```
one_valued =_def λA,B:Set, R:Rel A B. ∀a:A, b1,b2:B.
                    (R a b1) → (R a b2) → (b1 = b2).

pfunc arity =_def mk_pfunc
      { reln      : (Rel (vector nat arity) nat);
        One_valued: (one_valued (vector nat arity) nat reln)}.

pfuncs =_def Pfuncs: (n: nat) → (pfunc n) → pfuncs.
```

The type `pfunc` is a record where the field `reln` is a relation between vectors and natural numbers, and the field `One_valued` is a theorem stating that `reln` is single valued. It can be seen that this is a dependent record as the type of the second field depends on the value of the first field. The type `pfunc` can be seen in some sense as a subtype of `reln`, as objects of type `pfunc` are the objects of type `reln` which are proved to satisfy the property given by `One_valued`.

With this type system one can define the notion that a program computes a function by

```
∀n:nat, p:prog, f:(pfunc n). computes p n f
   =_def ∀v:(vector nat n), x:nat. exec p v x ⇔ reln n f v x
```

which is more compact and elegant than an equivalent HOL definition, since the information stored in types has to be specified as terms:

```
∀n:num, p:prog, f:pfunc. computes p n f =_def one_valued n f ∧
        ∀v:num list. length v = n ⇒
            ∀x:num. exec p v x ⇔ apply f v x
```

A mechanism which translates objects in a dependent type theory into HOL objects is illustrated in [13] and an extension of the HOL logic to cover quantification over types is proposed in [17].

3.2 Constant Definitions

Here we list the different mechanism by which constant definitions can be specified in Coq and in HOL.

Simple Definitions In HOL given a closed term $x : \tau$, a new constant $c : \tau$ can be introduced in the current theory by the primitive rule of constant definition which also yields the theorem $\vdash c = x$. Thus, while in the Calculus of Constructions constants are convertible with their defining terms, in HOL the interchangeability of c and x is justified by the above theorem, which needs to be used whenever c and x have to be substituted for each other in other theorems.

Specifications The second primitive rule which introduces constants in HOL theories is called the rule of constant specification. It introduces a constant $c\colon \tau$ obeying some property $P(c)$, if its existence can be shown by a theorem $\vdash \exists x.P(x)$. The theorem $\vdash P(c)$ is returned by the rule. Note that only the existence of some x is required, rather than the existence of a unique x, and nothing else can be inferred about c, apart from $P(c)$ (and anything which can be inferred from $P(c)$). There is no such rule in the Calculus of Constructions although any constructive proof of $\exists x\colon \tau.P(x)$ is actually a pair $(w\colon \tau, p\colon P(w))$ containing a term of type τ and a proof stating that this term satisfies P. The HOL manual [10] introduces a primitive inference rule for type specification as well but there is no implementation of this rule yet.

Recursive Definitions The definition of primitive recursive functions over a recursive type is justified in HOL by a theorem stating the principle of primitive recursion which can be automatically derived by the type definition package. A library for defining well-founded recursive functions, which in general requires user intervention for proving that a relation is well-formed, is also included in the HOL system [24]. In Coq, primitive recursive functions are defined by a fixpoint operator. The syntax of actually defining such functions implicitly in the Coq is very crude. However, a mechanism which allows function definitions in an ML like systax with pattern matching is provided in the `Gallina` language as a macro for specifying case expressions. This mechanism can also be used on the definition of functions over dependent types.

Inductive Definitions The CIC includes rules for inductive definitions and are thus inbuilt in Coq. The `Gallina` specification language provides constructs for introducing (possibly mutually) inductive definitions as well as tactics for reasoning about them. Inductive definitions can be used for introducing inductive types and sets as recursive datatypes (as seen in section 3) and also for inductively defined relations. Since the implementation of the CIC in Coq includes rules for coinductive types, support for coinductive and corecursive definitions and reasoning by coinduction is also provided.

The HOL system provides a number of packages for defining inductive relations, which include Melham's original package [16, 2], support for mutually inductive definitions [23] and the more recent implementation due to Harrison [12]. Besides providing a mechanism for specifying definitions these packages include ML functions for reasoning about them and for automating them. It is argued (for instance in [11]) that inductive definitions can be introduced earlier in the HOL system and a number of frequently used relations in existing theories (such as the inequalities on natural numbers) can be redefined inductively so that users can for instance apply the principle of rule induction on them, much in the same fashion that it is done by Coq users.

4 Theorem Proving

This section illustrates the different proof strategies by which users of the Coq and HOL systems perform the actual theorem proving.

4.1 Forward Proving

Forward theorem proving is performed in HOL by applying ML functions which return theorems. This is done in Coq by constructing terms whose type corresponds to theorems. However since HOL users have direct access to the meta-language, one can implement more elaborate inference rules for forward theorem proving than simple constructions of terms in Coq. In general, theorem proving in Coq is done in a backwards manner by applying tactics.

4.2 Backward Proving

Both theorem provers support interactive tactic based goal directed reasoning. Basically the required theorem is stated as a goal and the user applies tactics which break the goal into simpler subgoals until they can be proved directly. Tactics also provide a *justification* for the simplification of a goal into subgoals, which derives the goal as a theorem from derivations of the subgoals. A goal usually consists of the statement which is required to be proved together with a number of assumptions which a proof of the goal can use.

Backward proving is supported in HOL through an implementation of a *goal-stack* data structure which provides a number of operations (including specifying goals, applying tactics, moving around subgoals, etc.) as ML functions. Tactics and tacticals[2] are also ML functions and users can implement new tactics during theory development. On the other hand, Coq tactics and tacticals are provided as constructs of the `Gallina` language, and so are the operations on the internal goalstack. As a result, implementing a new tactic in Coq involves the non-trivial task of extending the `Gallina` language and in general Coq users tend to implement less tactics during theory development than HOL users do. Moreover, HOL users can also implement tactics 'on the fly' by combining different tactics, tacticals, and ML functions in general. For instance, the HOL tactic

[2] Tacticals are operations on tactics which produce tactics, for example, the tactical `then`, implemented in both HOL and Coq takes two tactics t_1 and t_2 and returns a tactic which when applied to a goal, it first applies t_1 and then applies t_2 on all the resulting subgoals.

```
REPEAT (STRIP_GOAL_THEN
  (fn t => if is_disj (concl t)
             then DISJ_CASES_TAC t
             else RULE_ASSUM_TAC (REWRITE_RULE [GSYM t])));
```

when applied to a goal of the form

$$\forall x_{11}, \ldots, x_{1n_1}.t_1 \Rightarrow \ldots \Rightarrow \forall x_{m1}, \ldots, x_{1n_m}.t_m \Rightarrow c$$

specialises all the quantified variables x_{ij} and strips the terms t_i from the goal; if t_i is a disjunction then the goal is broken down into two, each one having one of the term's disjuncts as an assumption. For each term t_i which is not a disjunction, the tactic rewrites all the assumptions with `GSYM` t_i which is the result of substituting all the subterms in t_i representing some equality $x = y$ with their symmetry $y = x$. Such a tactic is impossible to construct in Coq within a `Gallina` theorem proving session.

We also remark that HOL tactics are much more elaborate and numerous than Coq ones. One reason for this arises from the different nature of the Calculus of Inductive Constructions and the HOL logic. Since theorems in Coq are essentially types, tactics correspond to the different ways terms can be constructed and broken down (the introduction and elimination rules of the constructs). On the other hand, tactics in HOL have to be implemented using the much less powerful (and less general) primitive inference rules. Moreover, the powerful notion of convertible terms of CIC makes inference rules such as rewriting with the definitions and beta conversion unnecessary in Coq. However, tactics for unfolding definitions and changing a goal or assumption to a convertible one are also provided, both because it facilitates theorem proving and also because higher order unification is undecidable and user intervention may sometimes be essential.

The considerable difference between the number (and nature) of tactics in HOL and in Coq and the availability of a specification and proof language makes Coq an easier system to learn. New HOL users are faced with hundreds of inference rules and tactics to learn, and possibly a new programming language to master in order to be used effectively as a metalanguage. New Coq users need to learn how to use about fifty language constructs and most theory development can be done without the need of extending `Gallina`.

Finally we note that assumptions in Coq are named while in HOL they are not. This affects the way user of the systems use assumptions during the construction of a proof. Basically Coq users select the assumptions they need by their name while HOL users apply tactics which try to use all the assumptions. Nevertheless, HOL users can implement tactics which select a subset of, or a particular element from, the list of assumptions through filtering functions and other techniques discussed in [1]. However we stress that selecting an assumption simply by its name is definitely more straightforward than any such techniques. During the implementation of [26] the need of writing several filtering functions was sometimes tedious and overwhelming. Tactics which make use of all the assumptions can however be quite powerful and may save several repetitive proof steps. One can for instance consider the power of `ASM_REWRITE_TAC` in HOL which repetitively rewrites with all the assumptions, a number of theorems supplied by the user and a list of basic pre-proved theorems (such as $\vdash \forall A.\ \top \vee A = \top$.)

4.3 Automation

The HOL system is equipped with more decision procedures and automation tools than Coq. HOL (HOL90 version 9.1α) includes automation for rewriting (by a simple rewriting engine, an implementation of Knuth-Benedix completion, and a contextual rewriter), a tautology checker, semidecision procedures for first order reasoning (a tableaux prover with equality, and a model elimination based prover), a decision procedure for Presburger arithmetic and for real algebra, as well as an implementation of Nelson and Oppen's technique for combining decision procedures. Since most proofs in [26] are of a highly technical nature, the use of such decision procedures saved a lot of time and thinking about trivial proofs. The Coq system (version 6.1) provides tactics for tautology checking, decision procedures for intuitionistic direct predicate calculus, for Presburger arithmetic, and for a number of problems concerning Abelian rings. The `Gallina` language contains also a user definable hint list, where tactics can be included into the list and goals can then be automatically solved by the application of one or more of these tactics.

4.4 Reasoning with Equality and Equivalence

HOL's notion of equality is extremely powerful and since equivalence of propositions is defined as equality on boolean values, the same properties enjoyed by equality hold also for equivalence. Equality is introduced in HOL by a primitive rule, `REFL`, which returns the theorem $\vdash t = t$ for any term t; and the primitive rule of substitution allows any subterms of a theorem to be substituted by their equals. The rule of extensionality (which can be derived in HOL) yields the equality of any two functions which give the same results when applied to the same values. (More formally, the rule of extensionality is $\forall x.f(x) = g(x) \vdash f = g$.) As a result, equivalent predicates can be substituted for each other and assumptions can be substituted with the truth value $\top$. Hence, theorem proving in HOL can rely a lot on rewriting, for example, statements like $a \wedge b \Rightarrow a \vee c$ can be easily proved by the tactic:

```
REPEAT STRIP_TAC THEN
ASM_REWRITE_TAC []
```

The importance of equality in HOL theorem proving is emphasized by a class of inference rules called *conversions* which are specialised for deriving equalities. Basically, a conversion is an ML function which takes a term, t_1, and proves that it is equal to some other term t_2 deriving $t_1 = t_2$.[3] Conversions can be used for

[3] Note that the term t_2 is constructed by the conversion and not given by the user. The use of a particular conversion is actually the transformation of the term t_1 into some term t_2 justified by the theorem $\vdash t_1 = t_2$.

instance to simplify a term based by rewriting with a particular definition, or to transform a term based on some calculation such as natural number arithmetic or reduction into conjunctive normal form. In general, conversions form the building blocks of more powerful automation tools.

Equality in CIC is introduced by the inductive definition

$$\texttt{Eq}\ A\ =_{def}\ \texttt{refl\_equal:}\ \forall a{:}A.\ \texttt{(eq}\ A\ x\ x\texttt{)}$$

and results like symmetry, transitivity and congruence can then be derived. However functions are intensional and equivalence of propositions is different from their equality. Basically, two propositions, a and b, can be proved to be equivalent in Coq by constructing a term with type $(a \rightarrow b, b \rightarrow a)$ and little support is given for taking advantage of the symmetric nature of bi-implication. The need for a more powerful support of equality is reduced by having the notion of convertible terms. However, here we remark on the inability of constructing a term $t{:}\,T_1$ directly, where t has type T_2 which is not convertible with T_1 and it can be proved that T_1 and T_2 are equal. For example, given some term v: `(vector nat` $(n+m)$`)`, then one cannot specify v as having type `vector nat` $(m+n)$ even though $(n+m)$ and $(m+n)$ are equal. This problem is encountered in [27] and for this particular example it is solved by defining a function `Change_arity`, such that, given a vector v: `(vector` A n`)` and a proof t of $(n = m)$, then `Change_arity` n m t A v has type `(vector` A m`)`:

```
Change_arity
  =def λn,m:nat, t:(n = m), A: Set, v: (vector A n).
      eq_rec nat n (vector A) v m t).
```

and it is proved that:

```
∀n:nat, t:(n = n), A:Set, v:(vector A n).
      Change_arity n n t A v = v
```

This theorem is proved using the `eq_rec_eq` axiom.

Now, if `plus_sym` represents the theorem $\forall n, m.n+m = m+n$, and the term v has type `vector nat` $(n+m)$ then

```
Change_arity (n + m) (m + n) (plus_sym n m) nat v
```

has type `vector nat` $(m+n)$.

5 Miscellaneous

This section lists some other considerations of the differences between the approaches of Coq and HOL to the mechanisation of theories.

5.1 Classical and Constructive reasoning

HOL's logic is classical, and the axiom of the excluded middle is introduced in the HOL theory which defines boolean values. One can ask however whether any support can be given to users who may want to use HOL and still reason constructively. The CIC is essentially constructive in which the law of the excluded middle cannot be derived and all Coq functions have to be computable. However, one can still reason classically to some extent in Coq by loading a classical theory which specifies the law of the excluded middle as an axiom, although it should be stressed that this does not give Coq the full powers of classical reasoning.

Since all functions in Coq are computable, n-ary partial functions in [27] are specified as single valued relations (see section 3.1) rather than as Coq functions, so that functions which are not computable can still be specified in the mechanisation. On the other hand, functions in HOL need not be computable (since the logic is not constructive and because of the rule of constant specification and Hilbert's operator ϵ), and n-ary partial functions in [26] are defined as HOL functions mapping lists of natural numbers to possibly undefined natural numbers. The type of possibly undefined numbers is defined as the type of natural numbers together with an undefined value. The advantage of the formalisation of partial functions in HOL is that a function application can be directly substituted by its value.

5.2 The Use of Proof Objects

The Coq system stores proof terms in its theory files and uses for these terms include:

1. Program extraction: Given some program specification S, a constructive proof that there is some program satisfying it contains an instance of a program for which S holds, hence one can obtain a certified program from a proof of its specification. This facility is supported by the Coq system which provides a package which extracts an ML program from a proof term, as well as providing support for proving the specification of functions written in an ML syntax [20, 19, 21].
2. Extracting proof texts written in a natural language: A proof term of type τ can be seen as an account of the proof steps involved in deriving the theorem τ, and Coq provides tools for extracting a proof written in a natural language from proof objects [6].
3. Independent proof checking: Proof terms can be checked by an independent proof checker to gain more confidence in their correctness. Moreover, such proof terms can be easier to translate into proof accounts of another theorem prover than an actual proof script or an ML program (as HOL proof scripts actually are). The HOL system is truth based rather than proof based and it does not store proofs in its theories.

5.3 The Sectioning Mechanism

The `Gallina` specification language allows Coq proof scripts to be structured into sections, and one can make definitions and prove theorems which are local to a particular section. The need of local definitions and results is often encountered during theory development, where for instance, the definition of some particular concept can facilitate the proof of a number of results but does not contribute much to the overall formalization of the theory. This point is also discussed in [26] where the following example is given. During the proof of the theorem stating that primitive recursive functions are URM computable, a program P, say, which computes some particular function is defined. This program can be broken down into three subroutines: P_1, P_2 and P_3. A number of lemmas concerning these subroutines are derived and used in the proof of the required theorem. However the definitions of P_1, P_2 and P_3 as well as any results concerning them are used only during the proof of one important theorem, and the lack of structure in HOL theories resulted in having to represent them as local variables within the metalangauge.

6 Conclusions

The two case studies, and especially more extensive mechanisations of different mathematical theories, show that both HOL and Coq are robust systems and practical in mechanising simple mathematical results. The strongest point of HOL is the flexibility given to the users by means of the metalanguage; while Coq theorem proving relies on the power of the Calculus of Inductive Constructions. Here, we give some concluding remarks on these features.

6.1 The Flexibility of the Metalanguage

By allowing a theorem proving session to be given within a general purpose metalanguage, HOL offers a higher degree of flexibility than Coq. As a result, HOL users implement a larger number of new inference rules during theory development than Coq users. For example, the mechanisation of the theory of computation in HOL includes several conversions for animating the definitions, simple and more elaborate tactics which avoid repetitive inferences and most backward proofs include tactics implemented 'on the fly'. The syntax of `Gallina` can be extended, say with (metalanguage) predicates on terms so that one can filter a sublist of assumptions to be used by some tactic, but then one asks whether a specification language as powerful as the metalanguage is required to implement the required filtering functions during theorem proving. Having a specification language surely has its advantages: the system is easier to learn by new users, and proof scripts are in general easier to follow; also, theorem proving support tools like a debugger or a graphical user interface are probably easier to develop for a specification language with a limited syntax rather than for a general purpose programming language. However, the power of a Turing complete metalanguage is not to be underestimated, for it can be used for instance to derive theorems through the manipulation of proof terms.

6.2 The Power of the Calculus of Inductive Constructions

The restrictions due to the specification language are relieved by the power of CIC. The fact that theorems are proved by simply constructing and breaking down terms makes the implementation of tactics specialised for particular logic constructs unnecessary and the powerful notion of convertibility replaces the implementation of conversions for every definition. No new tactics or inference rules are implemented in the mechanisation of the theory of computation in Coq, both because the inference power of the simple constructs of `Gallina` is enough for most reasoning, and also because the non-trivial task of actually implementing a new elaborate tactic in Coq discourages the development of simple tactics which are used only to substitute a number of inferences. The power of CIC is also emphasised by its highly expressive type system which allows quantification over types and dependent types and thus gives a more natural formalisation of mathematical concepts than a simple type theory. We have seen however, how the stronger notion of equality and equivalence in HOL simplifies most formalisations.

The primitive inference rules of HOL are too simple and are rarely used in practice, most reasoning is performed by higher level inferences. The simplicity of the primitive rules gives a straightforward implementation of the core inference engine, on whose correctness the soundness of the HOL system relies. Although CIC is more complex than the HOL logic, it is sound and due to the Curry-Howard isomorphism theorems in CIC can be checked by a type checking algorithm, on whose correctness the soundness of the Coq system relies. Thus, one can have a very powerful logic whose theorems can still be checked by a simple algorithm.

The feasibility of actually doing so may however be questioned. Proof terms may become very large, and $\beta\delta\iota$-convertibility may become infeasible for large objects. These factors do not yield any significant problems for the mechanisation of the results in [27] but may make Coq unsuitable for large scale 'real-world' theorem proving required by the industry.

7 Acknowledgements

I would like to thank my supervisor, Simon Thompson, for his support and encouragement as well as the anonymous referees for their comments and suggestions on an earlier draft of this paper.

References

1. P. E. Black and P. J. Windley. Automatically synthesized term denotation predicates: A proof aid. In E. T. Schubert, P. J. Windley, and J. Alves-Foss, editors, *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, volume 971 of *Lecture Notes in Computer Science*, pages 46–57, Aspen Grove, UT, USA, September 1995. Springer-Verlag.

2. Juanito Camilleri and Tom Melham. Reasoning with inductively defined relations in the HOL theorem prover. Technical Report 265, University of Cambridge Computer Laboratory, August 1992.
3. Alonzo Church. A formulation of a simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
4. Thierry Coquand and Gérard Huet. The calculus of constructions. Rapport de Recherche 530, INRIA, Rocquencourt, France, May 1986.
5. C. Cornes et al. The Coq Proof Assistant Reference Manual, Version 5.10. Rapport technique RT-0177, INRIA, 1995.
6. Yann Coscoy, Gilles Kahn, and Laurent Théry. Extracting text from proofs. Rapport de Recherche 2459, INRIA, Sophia-Antipolis Cedex, France, January 1995.
7. N.J. Cutland. *Computability: An introduction to recursive function theory.* Cambridge University Press, 1980.
8. J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithétique d'ordre supérieur.* PhD thesis, Université Paris VII, 1972.
9. Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science.* Springer-Verlag, 1979.
10. M.J.C. Gordon and T.F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic.* Cambridge University Press, 1993.
11. John Harrison. HOL done right. Unpublished Draft, August 1995.
12. John Harrison. Inductive definitions: Automation and application. In E. T. Schubert, P. J. Windley, and J. Alves-Foss, editors, *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, volume 971 of *Lecture Notes in Computer Science*, pages 200–213, Aspen Grove, UT, USA, September 1995. Springer-Verlag.
13. Bart Jacobs and Tom Melham. Translating dependent type theory into higher order logic. In *TLCA '93 International Conference on Typed Lambda Calculi and Applications, Utrecht, 16–18 March 1993*, volume 664 of *Lecture Notes in Computer Science*, pages 209–229. Springer-Verlag, 1993.
14. Per Martin-Löf. *Intuitionistic Type Theory.* Bibioplois, Napoli, 1984. Notes of Giovanni Sambin on a series of lectues given in Padova.
15. T.F. Melham. Using recursive types to reason about hardware and higher order logic. In G.J. Milne, editor, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 27–50, Glasgow, Scotland, July 1988. IFIP WG 10.2, North-Holland.
16. T.F. Melham. A package for inductive relation definitions in HOL. In M. Archer, J.J. Joyce, K.N. Levitt, and P.J. Windley, editors, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 350–357, Davis, California, August 1991. IEEE Computer Society, ACM SIGDA, IEEE Computer Society Press.
17. T.F. Melham. The HOL logic extended with quantification over type variables. In L.J.M. Claesen and M.J.C. Gordon, editors, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 3–18, Leuven, Belgium, September 1992. IFIP TC10/WG10.2, North-Holland. IFIP Transactions.
18. Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf type theory: an introduction.* Clarendon, 1990.
19. C. Parent. Developing certified programs in the system Coq - the Program tactic. In H. Barendregt and T. Nipkow, editors, *International Workshop on Types for*

Proofs and Programs, volume 806 of *Lecture Notes in Computer Science*, pages 291–312. Springer-Verlag, May 1993.

20. C. Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In Association for Computing Machinery, editor, *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, Austin, January 1989.
21. C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15(5-6):607–640, ?? 1993.
22. Robert Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1994.
23. R. E. O. Roxas. A HOL package for reasoning about relations defined by mutual induction. In J. J. Joyce and C.-J. H. Seger, editors, *Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications (HUG'93)*, volume 780 of *Lecture Notes in Computer Science*, pages 129–140, Vancouver, B.C., Canada, August 1993. Springer-Verlag, 1994.
24. K. Slind. Function definition in higher-order logic. In J. von Wright, J. Grundy, and J. Harrison, editors, *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'96)*, volume 1125 of *Lecture Notes in Computer Science*, pages 381–397, Turku, Finland, August 1996. Springer.
25. Simon Thompson. *Type Theory and Functional Programming.* Addison-Wesley, 1991.
26. Vincent Zammit. A mechanisation of computability theory in HOL. In *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics*, volume 1125 of *Lecture Notes in Computer Science*, pages 431–446, Turku, Finland, August 1996. Springer-Verlag.
27. Vincent Zammit. A proof of the S_n^m theorem in Coq. Technical Report 9-97, The Computing Laboratory, The University of Kent at Canterbury, 1997.