Classical Tools for Constructive Proof Search

James L. Caldwell* and Judith Underwood†

June 25, 1996

Abstract

In this paper, we compare program development by extraction from a constructive proof with direct verification of a function. Motivated by the development of a decision procedure for intuitionistic propositional logic, we have taken first steps toward considering how classical type systems might be used in support of constructive systems. To this end we explore two statements of theorems asserting the existence of the tableau decision procedure. The constructive proof of one has a tableau decision procedure as its extraction; the other, if proved classically by providing a explicit witness, results in a verified decision procedure. The systems considered are the classical type theory of the PVS system and the constructive type theory of the Nuprl system. Since lambda terms encode constructive proofs, the question of what the explicit witness of the PVS theorem might prove on the constructive side is also considered.

1 Introduction and Motivation

One powerful motivation for working in constructive type theory is that a constructive proof of the existence of an object can be interpreted as a program which computes such an object. This method of constructing a program ensures the result is correct as long as the system is sound. However, there are disadvantages to an approach which uses constructive logic exclusively. Parts of a constructive proof may be irrelevant to the actual computation which results. Although it is possible to reason classically in a constructive system, such reasoning is not generally well supported.

For these reasons, we may wish to use a proof tool for classical logic as an aid to developing a program via a proof in constructive type theory. How can we do this while still retaining the computational meaning of the proof? In this paper,

*Address: Mail Stop 130, 1 South Wright Street, NASA Langley Research Center, Hampton, VA 23681-0001, USA. Email: `jlc@air16.larc.nasa.gov`

†Address: Department of Computer Science, University of Edinburgh, The King's Buildings, Edinburgh EH9 3JZ, United Kingdom. Email: `jlu@dcs.ed.ac.uk`

we outline some pragmatic approaches to this problem. We do not attempt to develop a formal semantic connection between classical and constructive type theories but instead focus on the practical application of these ideas. Some steps have been taken towards such a connection. Howe has developed a modified semantics for Nuprl to allow HOL theorems to be used within Nuprl proofs [6]. Coq's Program tactic, described in [11, 12], provides automatic assistance for program verification. The approach described there can be viewed as the inverse of the program extraction process and is related to the discussion below in section 3.1. Another comparison of a constructive proof with its classical counterpart can be found in [1].

Translation techniques connecting classical logic with constructive logic have a long history (see [8] for a survey). These techniques provide a method for interpreting an arbitrary classical formula ϕ in a constructive system such that ϕ is provable classically if and only if its translation is provable constructively. In this case, however, we consider only particular kinds of classical proofs, namely, program verification results. We interpret these proofs as type inhabitation judgements in type theory. If we view an inhabitant of a type (the verified program) as a proof of the corresponding proposition (its specification), this interpretation gives us a new way of building constructive proofs.

Thus, the primary motivations for this work come from the possibility of applying classical tools in constructive proof developments. There is also motivation from the classical side, since the identification of an executable subset of the classical theory is a step towards providing capability to execute specifications as prototypes.

In the remainder of this paper, we illustrate our ideas by considering a particular example of a theorem with useful computational content, and show how the verification of a program in a classical setting might be interpreted as a proof in constructive type theory. The classical language we use is that of PVS, a specification language and verification system based on simply typed higher-order logic [9, 10]. As a constructive system, we will work mainly with the type theory of Nuprl [4, 7], but we shall also consider what properties are needed in general for the interpretation of verification results.

2 From extraction to verification and back

If the goal of a constructive proof development is the extraction of a program, it is often the case that parts of the proof are irrelevant to the computation which results. For example, a proof by induction may require a complex measure to demonstrate that the induction is well-founded, but this measure may be otherwise unused in the computation. In such a case, there is no loss of program correctness if the proof that the induction is well-founded uses classical logic. Similarly, restrictions on types may guarantee that the computation has some intended behavior, but do not affect how the calculation is performed. In some constructive type theories we can identify such properties by using subset types, whose elements must satisfy a given predicate; however, to show that a term has such a type, the proof that the predicate holds must still be constructive.

In such cases, the development of a constructive proof is more complex than is required for the purpose of program extraction. Furthermore, the program extract from a fully constructive proof will be extensionally equivalent to the extract from a primarily constructive proof which uses non-constructive techniques for non-computational aspects of the proof. Thus, we can conceivably develop a constructive type theoretic proof by combining a verification of a program using classical logic with a meta-theorem describing when properties proved classically may be interpreted constructively.

In the remainder of this section, we focus on the first of these two steps. We will begin by describing an example of a constructive theorem with useful computational content but whose proof also contains much which is irrelevant to the computation. Given a type-theoretic formalization of this theorem, we show how to transform it into a classical program verification problem. In the next section, we discuss in more detail what is required to re-interpret the resulting classically verified program as a constructive proof.

2.1 Example: a decision procedure as proof extraction

The example we consider is a decision procedure for intuitionistic propositional logic. Verified decision procedures provide sound ways of increasing the automation of theorem provers; the decision procedure we describe could even be used as a proof search tool in a system which permitted reflection of proofs [3, 2].

This decision procedure is a form of the tableau algorithm [5, 13] which represents the computational content of a constructive proof of the completeness of Kripke models for intuitionistic propositional logic. The completeness theorem says, in effect, "for any formula ϕ, there is either a proof of ϕ or a Kripke model in which ϕ is not forced." (We call such a Kripke model a *countermodel*.) When formalized in constructive logic, the "or" in the conclusion is naturally constructive, so that for a given ϕ, either a proof or a countermodel is actually constructed. We shall need to make this property explicit in a classical statement of the theorem.

In any system, we would need to formalize the notion of a proof of ϕ and a Kripke countermodel for ϕ. These can be defined as datatypes in any sufficiently rich system, and, for our current purposes, the details are mostly unimportant. We will make some assumptions for ease of exposition. First, we assume that Kripke models have a finite set of states and a decidable forcing relation. For proofs, we assume that, given a proof P and a formula ϕ, there is a decidable predicate which determines if P proves ϕ.

2.2 Formalizing the theorem

This version of the completeness theorem was originally formulated for Nuprl's constructive type theory. We will first consider this constructive statement of the theorem, then examine how it may be restated in a classical setting and the possible advantages this may bring. The effect of this restatement is to turn a search for a constructive proof into a program verification problem.

The constructive statement is:

$$\begin{array}{l} \forall \phi : \mathit{Formula} \\ \quad (\exists P : \mathit{Proof}\,.\mathit{Proves}(P, \phi)) \\ \quad \vee \\ \quad (\exists K : \mathit{Kripke\_model}\ \exists s : \mathit{states}\ (K).\mathit{not\ forces}\ (K, \phi, s)) \end{array}$$

The tableau algorithm for intuitionistic propositional logic is more complex than that for classical propositional logic, but the general procedure is the same. Given a formula ϕ, the algorithm begins by assuming the formula is false and building a tableau which describes the logical consequences of this. If these consequences prove contradictory, then a sequent proof can be extracted from the tableau; otherwise, a Kripke model and a state in that model at which the formula does not hold can be extracted from the non-contradictory parts of this tableau.

The proof of the theorem above is essentially by induction on the number of formulas which can be added to a tableau rooted at F ϕ. Eventually, no new logical consequences can be added to the tableau, and we can determine whether we have found a proof or a countermodel.

The essence of the computation, then, is a function from formulas to the disjoint union of proofs and Kripke models. The classical statement of the theorem is an assertion that such a function exists. (We consider the computability of this function later.) In the statement above, the disjoint union of proofs and Kripke models is implicit in the use of constructive disjunction. In a classical setting, we will have to make this explicit.

Given a disjoint union type constructor, predicates `inl?` and `inr?` for recognizing left and right injections of elements of this type, and functions `case_left` and `case_right` for extracting the injected elements, we produce the following formalization in PVS[1]:

```
completeness : THEOREM
 EXISTS (f:[Formula -> disjoint_union[Kripke,Proof]]) :
   FORALL (w:Formula) :
    (inr?(f(w)) => Proves(case_right(f(w)),w))
   AND
    (inl?(f(w)) => EXISTS (s:States(case_left(f(w)))) :
                             NOT forces(case_left(f(w)),w,s))
```

[1] We use `typewriter` font for statements in the language of PVS and use *italics* for statements in the constructive system.

We will later make one further refinement to this statement, once we have considered when a classical proof of it actually provides constructive information.

3 Using classical theorems constructively

In the previous section, we restated a constructive theorem as a classical theorem asserting the existence of a function f with a particular property. If we then prove this theorem, under what circumstances can this proof be used to help develop a constructive proof of the original theorem? Foremost, if we wish to interpret the function f as a proof, we must have proven the existential statement by supplying a concrete witness, rather than by contradiction. So, assuming we have some explicit description of a function, we ask, "If we interpret this function as a proof, what is it a proof of?"

We continue to concentrate on PVS as a classical system. Our goal is now a method for interpreting inhabitants of a PVS function type `{f:A->B | P(f)}` as constructive proofs. (Note that the existence of a function in the type `{f:A->B | P(f)}` is equivalent to the existence of `f:A->B` such that `P(f)`) There are two main questions to be answered:

- When does an inhabitant of a PVS function type actually correspond to a constructive proof?
- Given some PVS function which does correspond to a constructive proof, what is it a proof of?

It is out of the scope of this paper to give complete answers to these questions; however, in this section we outline some partial solutions.

3.1 PVS functions as Nuprl proofs

PVS has a rich language of expressions, including the usual arithmetic and logical operators. However, some aspects of this language are not computational; for example, quantifiers are allowed in boolean expressions, which permits expressions of type Bool such as

```
FORALL (a:nat) : ((a > 2) AND even?(a)) IMPLIES
  EXISTS (b,c:nat) : prime?(b) AND prime?(c) AND a = b + c
```

This expression describes Goldbach's conjecture, and its truth value is unknown. In particular, we cannot reduce this expression to a canonical value (i.e. true or false) of boolean type, hence it does not correspond to a valid Nuprl boolean expression.

We will not attempt a complete characterization of the computable portion of the PVS expression language. However, we observe that many PVS expressions are in fact computable, and a large class of these can be described informally as the type-correct terms built from:

- variables of type `Bool, int, nat` ...
- arithmetic operators (`+, *, -`)
- basic logical operators (`AND, OR, NOT`)
- boolean conditionals (`IF` ... `THEN` ... `ELSE` ...)
- lambda-binding and application
- type-checked `RECURSIVE` functions

Since PVS recursively defined functions are provided with a measure, which is reduced at each recursive call, such functions can be expressed in Nuprl's language using a fixpoint combinator. In both systems, type-checking such a function (which may require theorem proving) guarantees its termination. We will call the class of expressions constructed from the constructs listed above *computable*.

Of course, the different systems use different syntax and different representations for terms, so some translation is required, *e.g.* PVS specifications are parsable strings while Nuprl maintains editable structured forms. Also, the version of the lambda-calculus supported by PVS is a typed system a-la Church, while the underlying computational system of Nuprl is untyped. We do not consider the technicalities of the translation here but assume a natural translation from computable PVS expressions to extensionally equivalent Nuprl terms exists. This assumption is justified in part because function equality in both systems is extensional. Below, we will indicate the natural translation by change of font from `typewriter` font to *italics*.

Any PVS computable expression corresponds in a natural way to some proof in constructive type theory, since it is essentially a well-typed term in an extended lambda calculus. At the simplest level, then, we can extract a type such as `nat -> nat -> bool`. However, given a computable PVS expression, it may have a PVS type which is much more informative that the simple type inferable from the expression itself. For example, we may have an element of

```
{f:nat->nat| FORALL (x:nat) : (f(x) * f(x) <= x ) AND
                                 (x < (f(x)+1) * (f(x)+1))}
```

which guarantees the function computes the integer square root of its argument. Much of the expressiveness of PVS arises from the use of predicates to define subtypes of a type `T`. In PVS, these predicates can be any function from `T` to `Bool`; since they need not be decidable predicates, not all such subtypes will correspond to a constructive type.

3.2 PVS predicates as Nuprl types

The problem of determining in general when a PVS subtype defined by a predicate corresponds to a type in constructive type theory is complex, and the appropriate definition of such a correspondence is open to debate. identifying computable expressions in PVS, some partial solutions are available. We will first consider some general principles, then describe pragmatically motivated restrictions suited to interpretations of PVS subtypes.

In order to produce an element of a set type $\{x : T \mid P(x)\}$ in constructive type theory, we would need to have a constructive proof of $P(x)$. Any mechanical translation from a classical interpretation to a constructive one would need to build this proof automatically. For this to work, the predicate P must be (constructively) decidable. If, instead, we are given a particular t of type T and wish only to verify that t is in the constructive type $\{x : T \mid P(x)\}$, then we only need that $P(t)$ is constructively decidable, a generally weaker condition.

Although there are some general syntactic classes of decidable predicates, we instead exploit the fact that PVS predicates are functions to booleans. If a predicate is actually a *computable* function, then not only do we know that the predicate is decidable, but we are given the means of producing a constructively valid element of *bool* which describes the truth value of the predicate. This means we preserve the classical nature of the connectives in the predicate; since we assume these predicates are decidable, their intuitionistic truth values should agree with their classical truth values. We assume we can lift the boolean truth value to a canonically true or false constructive proposition with a function *bool_to_prop* .

Our initial rule is then:

> If `t` is a computable PVS expression in the PVS type `{x:T | P(x)}`, then t is in the constructive type $\{x : T \mid \mathit{bool\_to\_prop}\ (P(x))\}$ if $P(t)$ is a computable boolean expression.

In fact, we can extend this rule if `P(x)` is a universally quantified PVS formula. For example, if we have `P(x) = FORALL (y:S): P'(x,y)`, where `P'(x,y)` is a computable PVS expression of type Bool, then we can use the term $\lambda y.P'(x, y)$. as a proof of $S \rightarrow \mathit{bool}$. Thus, we can constructively interpret any prenex universal quantifiers using a dependent function type.

Using this idea, we can generalize the rule above to one including universal quantifiers. For convenience, we show only one:

> If `t` is a computable PVS expression in the PVS type `{x:T | FORALL (y:S): P'(x,y)}`, then t is in the constructive type $\{x : T \mid \Pi(y : S).\mathit{bool\_to\_prop}\ (P'(x, y))\}$ if $P'(x, y)$ is a computable boolean expression.

How does this interpretation apply to the example of the decision procedure as constructive extract? Recall that the PVS statement we had was:

```
completeness : THEOREM
  EXISTS (f:[Formula -> disjoint_union[Kripke,Proof]]) :
    FORALL (w:Formula) :
      (inr?(f(w)) => Proves(case_right(f(w)),w))
     AND
      (inl?(f(w)) => EXISTS (s:States(case_left(f(w)))) :
                                  NOT forces(case_left(f(w)),w,s))
```

Since we assumed a decidable forcing predicate, we can interpret the boolean predicate `forces` as a computable function. Similarly, we have assumed a constructive interpretation of `Proves` as well. The difficulty in the interpretation of the predicate on `f` lies only in the presence of the boolean quantifiers `FORALL (w:Formula)` and `EXISTS (s:States...)`. We outlined above how to interpret the initial universal quantifier. The existential is slightly more problematic and can not, with the interpretation defined so far, be automatically translated back into the constructive context. However, because the set of states of a Kripke model is assumed to be finite, we know (as a meta-theorem) that the existential quantifier is equivalent to some finite disjunction, and thus the body of the predicate could be made computable.

Based on these observations we eliminate the existential quantifier by recasting the theorem as follows:

```
completeness : THEOREM
  EXISTS (f:[Formula -> disjoint_union[Kripke,Proof]]) :
    FORALL (w:Formula) :
      (inr?(f(w)) => Proves(case_right(f(w)),w))
     AND
      (inl?(f(w)) => Countermodel(case_left(f(w)),w))
```

The role of the existential quantifier in the constructive statement was to provide evidence that the formula was not forced in the Kripke model. However, since we know the Kripke models are finite and, at each state, the forcing relation is decidable, checking if a model has a state which fails to forces a formula is also decidable. This provides a specification for the new predicate: `Countermodel(K,a)` is true implies the constructive existence of a state which does not force `a`; but, since the set of states in a model is finite, this is straightforward.

4 Conclusions

We have shown a statement of intuitionistic propositional decidability which has a constructive proof yielding a tableau decision procedure. We have shown a translation of the constructive theorem into a classical theorem which explicitly asserts the existence of a function implementing the decision procedure. We massaged the statement of the classical theorem into a form which guarantees the witness function provided to discharge the existential has a computable type. Finally, we suggested how a computable term provided as a witness function in a proof of the classical theorem might be reinterpreted as a constructive proof.

Formalization and mechanically checked proofs of the two completeness theorems presented here are currently underway, one in PVS and the other in Nuprl.

The relationship between the development of the two proofs eventually needs to be based on semantic agreement of the datatypes formalized in the two systems. The key datatypes that must agree are those for formula, Kripke models, and proofs. The extensional equivalence of the predicates for provability, countermodel, and forcing must also be established.

References

[1] Sten Agerholm, Ilya Beylin, and Peter Dybjer. A comparison of HOL and ALF formalizations of a catagorical coherence theorem. Computer Science Department, Chalmers University of Technology, Göteborg, Sweden. Manuscript available at `http://www.cs.chalmers.se/~peterd/`, March 1996.

[2] William Aitken, Robert Constable, and Judith Underwood. Metalogical frameworks II: Using reflected decision procedures. unpublished manuscript.

[3] D. Basin and R. Constable. Metalogical frameworks. In G. Huet and G. Plotkin, editors, *Logical Environments*, chapter 1, pages 1–29. Cambridge University Press, Great Britain, 1993.

[4] Robert L. Constable and et al. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, N.J., 1986.

[5] Melvin Fitting. *Intuitionistic Logic, Model Theory, and Forcing.* North-Holland, 1969.

[6] D. J. Howe. Semantic foundations for embedding HOL in Nuprl. In *Fifth International Conference on Algebraic Methodology and Software Technology AMAST '96*, July 1996. To appear as a volume in LNCS.

[7] Paul Jackson. The Nuprl proof developemnt system, version 4.2 reference manual and user's guide. Computer Science Department, Cornell University, Ithaca, N.Y. Manuscript available at `http://www.cs.cornell.edu/Info/Projects/NuPrl/manual/it.html`, July 1995.

[8] C. Murthy. *Extracting Constructive Content from Classical Proofs.* PhD thesis, Cornell University, Dept. of Computer Science, 1990. (TR 89-1151).

[9] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[10] Sam Owre and Natarajan Shankar. The formal semantics of PVS. Computer Science Laboratory, SRI International, Menlo Park, CA. Draft Manuscript available at `http://www.csl.sri.com/~shankar/shankar-drafts.html`, October 1995.

[11] C. Parent. Developing certified programs in the system Coq- The Program tactic. In H. Barendregt and T. Nipkow, editors, *Types For Proofs and Programs*, volume 806 of *LNCS*, pages 291–312, May 1993.

[12] C. Parent. Synthesizing proofs from programs in the Calculus of Inductive Constructions. In *Mathematics for Programs Constructions'95*, volume 947 of *LNCS*, July 1995.

[13] Judith Underwood. The tableau algorithm for intuitionistic propositional calculus as a constructive completeness proof. In *Proceedings of the Workshop on Theorem Proving with Analytic Tableaux, Marseille, France*, pages 245–248, 1993. Available as Technical Report MPI-I-93-213 Max-Planck-Institut für Informatik, Saarbrücken, Germany.