The Boyer-Moore Prover and Nuprl: An Experimental Comparison

David Basin
Department of Artificial Intelligence,
University of Edinburgh, Edinburgh Scotland.
Email: basin@aipna.ed.ac.uk

Matt Kaufmann
Computational Logic, Inc.
Austin, Texas 78703 USA
Email: kaufmann@cli.com[1]

Abstract

We use an example to compare the Boyer-Moore Theorem Prover and the Nuprl Proof Development System. The respective machine verifications of a version of Ramsey's theorem illustrate similarities and differences between the two systems. The proofs are compared using both quantitative and non-quantitative measures, and we examine difficulties in making such comparisons.

1 Introduction

Over the last 25 years, a large number of logics and systems have been devised for machine verified mathematical development. These systems vary significantly in many important ways, including: underlying philosophy, object-level logic, support for meta-level reasoning, support for automated proof construction, and user interface. A summary of some of these systems, along with a number of interesting comments about issues (such as differences in logics, proof power, theory construction, and styles of user interaction), may be found in Lindsay's article [14]. The Kemmerer study [13] compares the use of four software verification systems (all based on classical logic) on particular programs.

In this report we compare two interactive systems for proof development and checking: The Boyer-Moore Theorem Prover and the Nuprl Proof Development System. We have based our comparison on similar proofs of a specific theorem: the finite exponent two version of Ramsey's theorem (explained in Section 2). The Boyer-Moore Theorem Prover is a powerful (by current standards) heuristic theorem prover for a quantifier-free variant of first order Peano arithmetic with additional data types. Nuprl is a tactic-oriented theorem prover based on a sequent calculus formulation of a constructive type theory similar to Martin-Löf's [15]. We do not assume any prior knowledge of either system by the reader.

Throughout this paper, we refer to the Boyer-Moore Theorem Prover by its name, NQTHM. "NQTHM" stands for "New Quantified THeorem Prover", in reference to a quantification feature added in about 1984.

Why undertake such a comparison? We believe there is too little communication between those who use different mechanical proof-checking systems. This lack of communication encourages myths

[1]Earlier related work was supported by ONR Contract N00014-81-K-0634.

and preconceptions about the usability of various systems. Concrete comparisons bring to light the limitations and advantages of different systems, as well as their commonalities and differences. Moreover, comparisons help determine which directions the field is heading and what progress is being made.

Much of our comparison is based on our two proofs of the aforementioned theorem. We make qualitative comparisons, and we also make quantitative comparisons based on metrics that indicate the degree of effort required to prove the theorem. However, we caution that there is a real danger in quantitative comparisons as metrics can oversimply and mislead. Numbers cannot account for much of what is important about proof development systems and their application; e.g.,

- Expressive power of the underlying logic (e.g., first order vs. higher order, definitional extensibility, set theoretic or arithmetic, typed vs. untyped.)
- Domains particularly well-suited to verification with the given system
- Computational content (explicit or implicit), i.e., existence of executable programs associated with existence proofs (written explicitly or derived implicitly from the proof)
- Use of novel techniques in representation or in automated reasoning
- Naturalness/comprehensibility of definitions and proofs, both to experienced users and to a more general community
- Ease of creating, reviewing, and changing definitions and proofs
- Proof discovery capabilities
- Soundness

Moreover, the choice of metrics is difficult. Different metrics can favor different kinds of systems more than one might expect. For example, the number of tokens typed may not correlate with number of keystrokes if one system uses extensive cutting and pasting with an editor, or uses keyboard macros or structure editor facilities. For another example, it may or may not be the case that prover power correlates with replay time; it's not hard to envision scenarios in which a weak system's proofs replay more quickly because there is a large proof script but little heuristic search. Nonetheless, we make quantitative comparisons using the following metrics.

- Lemma and definition counts
- Symbol counts
- User time required
- Replay time

Our paper is organized as follows. Section 2 provides an informal statement and proof of the version of Ramsey's theorem alluded to above. Section 3 contains a description of the proof completed with the Boyer-Moore Theorem Prover, NQTHM. Section 4 contains the Nuprl proof. The final section draws comparisons and conclusions.

Acknowledgements. We thank Gian-Luigi Bellin, Jussi Ketonen, and David McAllester for many interesting and useful discussions on this topic. We thank Andrew Ireland, Bill Pierce, Matt Wilding, and the referees for their very helpful comments on a draft of this paper. Randy Pollack gave us some useful encouragement on this effort. David Basin also gratefully acknowledges assistance provided by Doug Howe during the Nuprl proof effort.

2 Informal Statement and Proof of Ramsey's Theorem

The version of Ramsey's theorem that we prove is about the existence of "order" in symmetric graphs. A *symmetric graph* is a set V of vertices together with a symmetric edge relation E on that set. (Some formulations also require that E is irreflexive; that bears little on the essential mathematics but can bear on the details.) Let G be a symmetric graph with vertex set V and relation E. A *clique* C of G is a subset of V such that all pairs $<x, y>$ of distinct members of C belong to E; an *independent set* I of G is a subset of V such that no pairs $<x, y>$ of members of I belong to E. A set is *homogeneous* if it is a clique or an independent set. For any positive integer l we say that a subset S of V is an *l-clique* (respectively, an *l-independent set*) if it is a clique (respectively, independent set) of cardinality l. Finally, for any positive integers n, l_1 and l_2 we write

$$n \rightarrow (l_1, l_2)$$

to assert that for every graph G with at least n vertices, there is either an l_1-clique or an l_2-independent set in G.

Note: Henceforth all our graphs will be symmetric, i.e., *graph* means *symmetric graph.*

We may now state the main theorem.

Theorem. For all l_1, l_2, there exists an n such that $n \rightarrow (l_1, l_2)$.

Note: The least such n is sometimes called the *Ramsey number* corresponding to l_1 and l_2.

An informal "text-book proof" (similar to one in [9]) proceeds by double induction on l_1 and l_2. (Alternatively, the proof may proceed by a single induction on their sum — this is the tact taken in the NQTHM proof.) To prove the base case observe that $l_1 \rightarrow (l_1, 2)$ as any graph with l_1 vertices either is a clique, or there are at least two vertices that are not connected by an edge. Similarly $l_2 \rightarrow (2, l_2)$. Now assume as an inductive hypothesis that we have some n and m where $n \rightarrow (l_1, l_2 - 1)$ and $m \rightarrow (l_1 - 1, l_2)$.

Claim. $n + m \rightarrow (l_1, l_2)$.

Proof. Given an arbitrary graph G on at least $n + m$ vertices, choose an element v_0 of its vertex set V. Now partition the remaining elements into two sets r_1 and r_2 where

$$\begin{aligned} r_1 &= \{x \in (V - \{v_0\}) : E(v_0, x)\} \\ r_2 &= \{x \in (V - \{v_0\}) : \neg E(v_0, x)\} \end{aligned}$$

Then $|r_1| + |r_2| \geq m + n - 1$ so either $|r_1| \geq m$ or $|r_2| \geq n$. If $|r_1| \geq m$, then by the induction hypothesis there is a subset S of r_1 where either S is an l_2-independent set (so we are done) or S is an $(l_1 - 1)$-clique. In the latter case set $S' = S + v_0$. Now, since $S \subseteq r_1$, there is an edge between v_0 and all $x \in S$. Hence S' is the desired l_1-clique. The case $|r_2| \geq n$ is analogous.

3 The Boyer-Moore Theorem Prover NQTHM

We present our discussion of the NQTHM proof in five subsections. We begin by giving enough of an introduction to the logic of NQTHM to be able to transform a straightforward first order formalization of the theorem into a formalization in the NQTHM logic (which we also do in the first subsection). Afterwards, we discuss the proof strategy. In the third subsection, we summarize the actual proof steps. We continue with some statistics and general remarks about the proof and we conclude with a discussion of computing in the NQTHM logic.

3.1 Background and Formalization of the Theorem

The Boyer-Moore theorem prover NQTHM is a heuristic prover based on a simple version of a traditional first order logic of total functions, with instantiation and induction rules of inference. The logic is quantifier-free, except for the implicit universal quantification surrounding each definition and theorem. A detailed description of the logic and manual for the prover may be found in [6]. The lack of quantifiers together with the presence of induction encourages a style of specification and proof that is "constructive" in nature. In fact, Bob Boyer has pointed out that since definitions in the NQTHM logic (technically speaking, in "thm mode", i.e., without the `V&C$` interpreter and without induction all the way up to ϵ_0) always produce primitive recursive functions, the law of the excluded middle is constructively valid for this logic when all function symbols are introduced with definitions (DEFN events). "Constructive" here also means that rather than specifying the existence of various objects, one explicitly defines functions that yield those objects. The system is equipped with a mechanism for computing with its defined functions (see subsection below entitled *Computing*).

The syntax of the logic is in the Lisp tradition, where the list `(f` x_1 ... x_n`)` denotes the application of the function `f` to the arguments x_1 through x_n. We also allow the `let` form from Lisp (as it is included in the "PC-NQTHM" system described below), so for example the expression `(let ((x` a`) (y` b`))` exp`)` is equivalent to the result of substituting a for `x` and b for `y` in exp. Certain functions are built into the logic, such as the ordered pair constructor `cons` and the atom `nil`. The list (x_1 x_2 ... x_n) is represented by the term `(cons` x_1 `(cons` x_2 ... `(cons` x_n `nil)` ... `))`. The functions `car` and `cdr` select (respectively) the first and second component of a pair. Thus `car` selects the first member of a list. It is an axiom of the logic that every pair x equals `(cons (car` x`) (cdr` x`))`. Note also that since this is a logic of total functions, it makes sense to form the term `(car` x`)` for any term x, whether it is a pair (or list) or not. In practice, this lack of typing is generally not much of a problem for users once they become accustomed to it.

Consider now how we might formalize Ramsey's Theorem in this logic. If we had quantifiers then we might simply write the following. Here `pairs` is a list of pairs that represents a graph on a set `domain` of nodes, and `S` is the desired homogeneous set (i.e., a clique or an independent set) with respect to the graph `pairs`.

```
     For all l1 and l2 there exists N
     such that for all domain and pairs there exists S
     such that:
     N ≤ cardinality(domain)  →
     [S ⊆ domain ∧
(a)   ((l1 ≤ cardinality(S) ∧ clique (pairs, S)) ∨
(b)    (l2 ≤ cardinality(S) ∧ independent (pairs, S)))]
```

In order to represent this conjecture in the NQTHM logic, we must first eliminate the quantifiers. To that end, we will define `N` as a function `ramsey` of `l1` and `l2`, and we will also define `S` in terms of a function `wit` ("witness") of `l1`, `l2`, `domain`, and `pairs`. Actually, it is convenient to define `wit` to return an ordered pair of the form $<S, f>$, where S is the desired clique or independent set according to whether the "flag" f is 1 or 2, respectively. We'll say that `good-hom-set (pairs, domain, l1, l2, flg)` holds iff `flg` is 1 and disjunct (a) above holds, or else `flg` is not 1 and disjunct (b) above holds, where `S` is the first component of `wit (pairs, domain, l1, l2)`. The conjecture above thus transforms into the following statement.

```
For all l1, l2, domain, pairs, if
    S = car (wit (pairs, domain, l1, l2)) and
    flg = cdr (wit (pairs, domain, l1, l2)), then:
        ramsey (l1,l2) ≤ cardinality(domain)
        →
        [S ⊆ domain ∧ good-hom-set (pairs, domain, l1, l2, flg)]
```

The NQTHM logic has a built-in theory of lists, but not of sets. Therefore it is convenient to recast this formalization in terms of lists. We can define a predicate `setp` for a list having no duplicates. Slipping into Lisp-style syntax, we finally obtain a formalization of Ramsey's theorem. (Only the last conjunct below is new, and it says that if `domain` represents a set then so does the witness set. This is important so that the `length` of a list equals the cardinality of the set it represents; note that `length` is used in the definition of `good-hom-set`, which in turn may be found in the subsection below entitled *Outline of Main Proof Steps*.)

```
THEOREM.
     (implies (leq (ramsey l1 l2)
                   (length domain))
              (let ((pair (wit pairs domain l1 l2)))
                (let ((hom-set (car pair))
                      (indep-or-clique (cdr pair)))
                  (and (subsetp hom-set domain)
                       (good-hom-set pairs domain l1 l2 indep-or-clique)
                       (implies (setp domain)
                                (setp hom-set))))))
```

3.2 Proof Strategy

In the NQTHM prover, a "proof" is a sequence of steps called *events*, typically of two kinds: definition (DEFN) events and theorem (PROVE-LEMMA) events. A major step is to define the function `wit`, referred to above, that constructs the clique or independent set. Unlike Nuprl, the system does not construct such a function from the proof; rather, the function is introduced by the user and its pattern of recursion is available for generating heuristic induction schemes. In fact, the proof is eventually (after some lemmas are proved) accomplished using an induction heuristically chosen by the NQTHM system (see [6] or [4] for more on this topic) to reflect the recursion in the definition of the function `wit` below, i.e., by an induction on `(plus l1 l2)`. This function returns a pair for the form `(cons set flag)` where `flag` is 1 or 2 according to whether `set` is a clique or an independent set. Everything to the right of any semicolon is a comment.

```
DEFINITION.
(wit pairs domain l1 l2) =
(if (listp domain)
    (if (zerop l1) (cons nil 1)
      (if (zerop l2) (cons nil 2)
        (let ((set1 (car (partition (car domain) (cdr domain) pairs)))
              (set2 (cdr (partition (car domain) (cdr domain) pairs))))
           ;; the function sub1  below is the decrement function on natural numbers
          (if (lessp (length set1) (ramsey (sub1 l1) l2))
            ;; then use set2 to form clique or independent set
            (let ((wit-set2 (wit pairs set2 l1 (sub1 l2))))
              (if (equal (cdr wit-set2) 1) wit-set2
                (cons (cons (car domain) (car wit-set2))
                      2)))
            ;; otherwise use set1 to form clique or independent set
            (let ((wit-set1 (wit pairs set1 (sub1 l1) l2)))
              (if (equal (cdr wit-set1) 2) wit-set1
                (cons (cons (car domain) (car wit-set1))
                      1)))))))
  (cons nil 1))
```

This definition is actually presented with a *hint* `(lessp (plus l1 l2))` that instructs the prover to verify that the sum of the final two arguments of `wit` decreases in each recursive call of `wit`, thus guaranteeing termination. This informal description of what the prover does with that hint reflects a formal definitional principle in the NQTHM logic, but we omit further discussion of this point.

In this example we use two main techniques to "discover" the proofs. One approach (the traditional one for NQTHM users) is to start by presenting a lemma to NQTHM. If the proof fails or if the output (a mixture of English and formulas) suggests that the proof probably won't complete successfully, then inspection of the output often suggests (to an experienced eye) useful rewrite (simplification) rules that one might wish to prove. The other main technique is to use an interactive enhancement [12] to the NQTHM system as an aid to discovering the structure of the proof. This "PC-NQTHM" enhancement ("PC" for "proof-checker") allows one to create PROVE-LEMMA events by first submitting the proposed theorem and then interactively giving various proof commands in a backward goal-directed, "refinement" style. These commands range from "low-level" commands which invoke a particular definition or rewrite rule, to "medium-level" commands invoking simplification or (heuristic) induction, to "high-level" commands which call the NQTHM prover. There is also a facility for user-defined *macro commands* in the tradition of the "tactics" and "tacticals" of LCF [8] and Nuprl (see Section 4). This "proof-checker" enhancement is helpful, but not crucial, for completion of the proof. The plan was in fact to develop nice rewrite rules rather than to rely on the manual commands provided by the interactive enhancement, and this plan succeeded: the final proof contained only 10 definitions and 26 lemmas (including the final theorem) after loading the standard "ground-zero" base theory, and did not contain any proof-checker commands. The events run successfully in the unenhanced NQTHM system. Other than a few hints – 7 of the lemmas took standard NQTHM "hints" that specify use of one or two previously proved lemmas – the proofs are fully automatic.

3.3 Outline of Main Proof Steps

We divide this subsection into four parts: one for the requisite definitions and then one for each of the three conjuncts of the conclusion of the main theorem (see subsection above entitled *Background and Formalization of the Theorem*).

3.3.1 Definitions

Several definitions are necessary for the proof. One of these is the definition of the function `wit`, provided in the preceding subsection, which picks out a desired clique or independent set. Another important definition is of a function `ramsey` that provides an upper bound on the Ramsey number. Here is that definition, expressed in the official syntax with formal parameters `l1` and `l2`.

```
(defn ramsey (l1 l2)
  (if (zerop l1) 1
      (if (zerop l2) 1
          (plus (ramsey (sub1 l1) l2)
                (ramsey l1 (sub1 l2)))))
  ((lessp (plus l1 l2))))
```

The last line of `ramsey` is a hint to the prover for its proof of termination and suggests that the sum of the two arguments decreases on each recursive call.

The definition of `wit` depends not only on the function `ramsey`, but also on three other functions, which we describe here informally; see [11] for details. For any list `pairs`, `(related i j pairs)` is *true* (`t`) if and only if the pair <i,j> or the pair <j,i> belongs to `pairs`; in this way we represent the notion of symmetric binary relation. `(partition n rest pairs)` returns a pair <x,y> where `x` consists of all elements of the list `rest` that are related (in the sense above) to `n`, and `y` consists of the remaining elements of `rest`. Finally, `(length lst)` is the length of the list `lst`.

The function `setp` recognizes whether or not a list represents a set by returning `t` if and only if the list contains no duplicates. Notice the use of primitive recursion to express bounded quantification. This is a common technique for defining universally-quantified concepts in the NQTHM logic.

```
(defn setp (x)
  (if (listp x)
      (and (not (member (car x) (cdr x))) (setp (cdr x)))
    t))
```

The function `homogeneous` defined below recognizes whether a given set `domain` is homogeneous (i.e., a clique or independent set) for the relation (graph) represented by the list `pairs`: it tests whether `domain` is a clique if the formal parameter `flg` is 1, and otherwise it tests whether `domain` is an independent set. Notice that `homogeneous` is defined by recursion on its first argument, using an auxiliary function `homogeneous1` that checks that its first parameter `n` is related or not related (according to whether or not the parameter `flg` is 1) to every element of `domain`.

```
(defn homogeneous1 (n domain pairs flg)
  (if (listp domain)
      (and (if (equal flg 1)
               (related n (car domain) pairs)
               (not (related n (car domain) pairs)))
           (homogeneous1 n (cdr domain) pairs flg))
      t))
```

```
(defn homogeneous (domain pairs flg)
  (if (listp domain)
      (and (homogeneous1 (car domain) (cdr domain) pairs flg)
           (homogeneous (cdr domain) pairs flg))
      t))
```

Our formalization of Ramsey's theorem also requires the notion of a "sufficiently large" homogeneous set, i.e., one that has at least `l1` or `l2` elements depending on whether the set is a clique or an independent set.

```
(defn good-hom-set (pairs domain l1 l2 flg)
  (and (homogeneous (car (wit pairs domain l1 l2))
                    pairs
                    flg)
       (not (lessp (length (car (wit pairs domain l1 l2)))
                   (if (equal flg 1) l1 l2)))))
```

Finally, we need the notion of subset. We omit here its straightforward recursive definition as well as several standard related lemmas such as transitivity.

3.3.2 The constructed set is contained in the given set

We wish to prove the following lemma. Notice the syntax for lemmas:

(prove-lemma *lemma-name* *lemma-types* *statement*),

where *lemma-types* is often `(rewrite)` to indicate that the lemma is to be used in subsequent proofs as a rewrite rule. (See [6] for details.)

```
(prove-lemma subsetp-hom-set-domain (rewrite)
  (subsetp (car (wit pairs domain l1 l2))
           domain))
```

The theorem prover proceeds by an induction suggested by the recursion in the definition of the function `wit`. The proof actually fails at first, but the prover's output suggests some lemmas.

Here is one of those, which says that the second component returned by `partition` is a subset of the given set.

```
(prove-lemma subsetp-cdr-partition (rewrite)
  (subsetp (cdr (partition x z pairs))
           z))
```

A similar lemma is required for the first component, and these are used to prove two rather technical lemmas suggested by the attempted proof of the lemma `subsetp-hom-set-domain`. After these four lemmas, the inductive proof of `subsetp-hom-set-domain` succeeds without further interaction.

3.3.3 The constructed set is homogeneous and large enough

The following lemma is at the heart of the theorem.

```
(prove-lemma wit-yields-good-hom-set (rewrite)
  (implies (not (lessp (length domain) (ramsey l1 l2)))
           (good-hom-set pairs domain l1 l2
                         (cdr (wit pairs domain l1 l2)))))
```

Unfortunately, the proof attempt does not succeed at first, so we must prove supporting lemmas. This time we use the proof-checker enhancement PC-NQTHM (see subsection above entitled *proof-strategy*) of NQTHM to explore the situation. Specifically, an INDUCT command is used to invoke a heuristic choice of induction scheme, a PROVE command is used in order to call the NQTHM prover to dispose of the three "base cases", the function `good-hom-set` is expanded in one of the remaining four (inductive) cases, a casesplit is performed to create 8 subgoals, and finally inspection of one of those subgoals suggests the following lemma.

```
(prove-lemma homogeneous1-subset (rewrite)
  (implies (and (subsetp x domain)
                (homogeneous1 elt domain pairs flg))
           (homogeneous1 elt x pairs flg)))
```

The NQTHM prover proves this lemma automatically. Manual application of this rule (in the proof-checker), to the subgoal referred to above, suggests another lemma, which is also proved automatically.

```
(prove-lemma homogeneous1-cdr-partition (rewrite)
  (homogeneous1 elt (cdr (partition elt dom pairs)) pairs 2))
```

Further attempts to prove subgoals inside the proof-checker, as well as attempts to prove the main goal `wit-yields-good-hom-set` in the context of the two lemmas displayed above (and others discovered subsequently), lead to some additional lemmas. One is completely analogous to `homogeneous1-cdr-partition` (displayed just above), but for the `car` of the partition in place of the `cdr`. Another (also proved automatically) asserts that the cardinality of a set equals the sum of the cardinalities of the two sets returned by `partition`. Still another asserts that `ramsey` always returns a positive integer. Four other technical lemmas seem necessary for the prover's rewriter to behave properly; they are omitted here. Finally, the proof of our goal `wit-yields-good-hom-set` (see above) succeeds.

3.3.4 The constructed set is indeed a set

The final goal is to prove that the homogeneous set is really a set.

```
(prove-lemma setp-hom-set (rewrite)
  (implies (setp domain)
           (setp (car (wit pairs domain l1 l2)))))
```

The NQTHM prover does not succeed in proving this automatically, but the output suggests the following useful (though obvious) lemma whose proof does succeed automatically. It states that the lists returned by `partition` are sets if the given list is a set.

```
(prove-lemma setp-partition (rewrite)
  (implies (setp x)
           (and (setp (car (partition a x pairs)))
                (setp (cdr (partition a x pairs))))))
```

Two technical lemmas discovered using the proof-checker suffice for concluding the proof of `setp-hom-set`.

3.4 More Statistics and General Remarks

The first time we formalized and proved this theorem was in January, 1987. The resulting proof was very ugly, as the use of the (then new) proof-checker enhancement was quite undisciplined. It seems best to find a good combination of elegant rewrite rules even when one has the capabilities offered by the proof-checker. Perhaps surprisingly, the "manual" proof takes longer to replay than the "heuristic" one reported here: total time was 219.7 seconds as opposed to 394.7 seconds for the older proof, both on a Sun 3/60 with 16 megabytes of main memory.

It took about 7 hours of human time to complete the current proof effort, which resulted in a list of events that is accepted by NQTHM (without the interactive enhancement). That number may be somewhat misleading since the new proof effort took advantage of the definitions and a few of the lemmas created in the earlier version. However, the more disciplined approach used in the final effort suggested changes that were made in the definitions, so it seems reasonable to guess that an effort from scratch would have taken not much longer than 7 hours anyhow. A more detailed annotated chronicle of this proof effort may be found in [11].

3.5 Computing

The NQTHM system provides a mechanism for computing values of variable-free terms in the logic. This mechanism of providing *executable counterparts* to defined functions is important as part of the theorem proving process. For one thing, it allows the fast replacement of variable-free terms by constants. This was an important consideration in, for example, the operating system kernel proof described in [3], where there were many large variable-free terms in the proof; compiling the executable counterparts sped up the proof process significantly.[2] Fast execution is also valuable when executing *metafunctions* [5], which are functions that are written in the logic for the purpose of simplifying terms and can be used as code once they are proved correct.

In this subsection we address the use of the NQTHM logic as a general purpose programming language. For example, if we want a homogeneous set of cardinality 3, execution of the function `ramsey` tells us that such a set exists within any graph with at least 20 vertices:

```
>(r-loop)  ;; We type this in order to enter the NQTHM reduction loop.
Trace Mode: Off   Abbreviated Output Mode:  On
Type ? for help.
*(ramsey 3 3)  ;; We ask the system to compute the value of  ramsey on inputs 3 and
3.....
 20
```

This is however an unsatisfactory answer, given that the Nuprl proof (see next section) provides a value of 6 rather than 20 in this case. Therefore, we have in fact re-done the proof using slightly different definitions of the functions `ramsey` and `wit` that more closely reflect the induction that takes place in the Nuprl proof, which is grounded at 2 rather than at 0. We then obtain 6 rather than 20 for the value of `(ramsey 3 3)`. It took roughly 5 hours to redo the proof for these new versions of `ramsey` and `wit`.

Convention. In the remainder of this subsection we refer to the alternate versions of `ramsey` and `wit` mentioned above.

The (new version of the) function `wit` can also be executed, on a particular set and binary relation, with particular values for its parameters `l1` and `l2`. Consider for example the hexagon, represented as a binary relation connecting i to $i+1$ for i from 1 to 6 (except that 6 is connected to 1).

```
*(wit '((1 . 2) (2 . 3) (3 . 4) (4 . 5) (5 . 6) (6 . 1))
      '(1 2 3 4 5 6)
      3 3)
 (CONS '(1 3 5) F)
```

Thus, the set $\{1, 3, 5\}$ is an independent set, as indicated by the second component `F` of the value returned above (which corresponds to `2` in the version discussed in the rest of this paper). We found on a Sun 3/60 that it took an average of about 0.15 seconds to evaluate this `wit` form. However, after compiling an appropriate file, the time was reduced to 0.011 seconds per form.

4 Nuprl

Our presentation of the Nuprl proof is divided into four subsections. The first provides a brief overview of Nuprl. The reader is encouraged to consult [7] for further details. The second summarizes definitions, theorems, and tactics used in our proof. The third presents main proof steps. And the final subsection documents the computational content of our proof.

[2] personal communication from Bill Bevier

4.1 Background

The basic objects of reasoning in Nuprl are types and members of types. The rules of Nuprl deal with *sequents*, objects of the form

$$x_1 : H_1,\ x_2 : H_2,\ \ldots,\ x_n : H_n >> A.$$

Informally, a sequent is true if, when given members x_i of type H_i (the *hypotheses*), one can construct a member (*inhabitant*) of A (the *goal* or *conclusion*). Nuprl's inference rules are applied in a top down fashion. That is, they allow us to refine a sequent obtaining subgoal sequents such that a goal inhabitant can be computed from subgoal inhabitants. Proofs in Nuprl are trees where each node has associated with it a sequent and a refinement rule. Children correspond to subgoals that result from refinement rule application.

These refinement rules may be either primitive inference rules or ML programs called tactics. Nuprl tactics are similar to those in LCF [8]: given a sequent as input, they apply primitive inference rules and other tactics to the proof tree. The unproved leaves of the resulting tree become the subgoals resulting from the tactic's application. Higher-order combinators called *tacticals* (e.g., *THEN* which sequences two tactics) are used to build complex tactics from simpler tactics. The correctness of the tactic mechanism is justified by the way the type structure of ML is used.

Nuprl's type theory is expressive; its intent is to facilitate the formalization of constructive mathematics. Higher order logic is represented via the propositions-as-types correspondence. Under this correspondence an intuitionistic proposition is identified with the type of its evidence or proof objects. For example, an intuitionistic proof of $A \Rightarrow B$ is a function mapping proofs of A to proofs of B, i.e., a member of the function space $A \rightarrow B$. Similarly, a proof of $A \& B$ inhabits the cartesian product type $A \# B$. A proposition is true when the corresponding type is inhabited. For example, $\lambda x.\, x$ is a member of the true proposition $A \Rightarrow A$. Types are stratified in an unbounded hierarchy of universes beginning with U_1. One may quantify over types belonging to a given universe, but the resulting type (predicatively) belongs to a higher universe.

To prove a proposition P of constructive mathematics in Nuprl, one proves "$>> P$" by applying refinement rules until no unproven subgoals exist. If P is true, a proof will produce a member (the proof object) that embodies the proof's computational content. Nuprl provides facilities to extract and execute this content. Thus, Nuprl may be viewed as a system for program synthesis: Theorem statements are program specifications, and the system extracts proven correct programs.

Theorem proving takes place within the context of a Nuprl *library*, an ordered collection of tactics, theorems, and definitions. Objects are created and modified using window-oriented, structure editors. Nuprl contains a definition facility for developing new notations in the form of templates (display forms) which can be invoked when entering text. Notations are defined using other definitions and ultimately terms within the type theory. Required properties (or axioms) about defined objects may not be assumed; they must be proved within the theory.

4.2 Theory Development

Our proof of Ramsey's theorem required approximately two weeks of work. Most of this time was spent building a library of foundations for reasoning about finite sets and graphs. Our self-contained library contains 24 definitions and 25 lemmas (counting the final theorem and excluding "ground zero" definitions such as the direct encodings of the logical connectives of predicate calculus).

Nineteen definitions and all of the lemmas are related to finite sets, four definitions are related to graphs, and one definition is relevant to the statement of Ramsey's theorem. A complete list of definitions and lemmas may be found in [1].

Our library is built in a rather general way and has been used by researchers at Cornell (in addition to the first author) to prove theorems in graph theory. Rather than assuming that finite sets are built from some specific type (such as integers) most of our definitions and theorems are parameterized by a type whose members are types with decidable member equalities. This type of types (which we refer to as the type of "discrete type pairs" and display as D) is defined as

$$T\!:\!U_1 \# \forall x, y : T.\, x = y \; in \; T \vee \neg(x = y \; in \; T).$$

For example, the type of integers (int) paired with a decision procedure for member equality belongs to this type D. If A is a member of D, we denote the first projection (i.e., the type or carrier) of A by $|A|$. Most definitions (e.g., finite set membership $\epsilon\{A\}$ or finite set subset $\subset \{A\}$) and theorems carry along a reference to this type parameter in their statement. Such generality is not required for Ramsey's theorem as we could have fixed a specific discrete type, such as the integers, throughout the proof; hence, in the presentation that follows, we often leave this parameter (A) implicit. A full discussion of the benefits of such parameterization may be found in [1].

Finite sets (displayed as $FS(A)$) are defined as lists without duplication, containing members from the carrier of some discrete type pair A. Specifically, finite sets are defined as the parameterized type[3]

$$\lambda A.\, \{l\!:\!|A| \; list \mid [nil \rightarrow True; h.t, v \rightarrow \neg(h \,\epsilon\{|A|\}\, t) \;\&\; v;\; @l]\}.$$

whose members are empty lists (the base case, when l is the empty list, evaluates to $True$), and all non-empty $|A|$ lists whose heads are not members of their tails and the same is recursively required of their tails. Given this definition, finite set membership (displayed as $\epsilon\{A\}$) is defined in the obvious way as list membership, subset (displayed as $\subset \{A\}$) in terms of finite set membership, cardinality as list length, etc.

Many of the theorems we prove about finite sets read like set theory axioms, and their proofs implicitly construct useful functions on finite sets. For example, the following lemma states that the union of two finite sets is a finite set: "for all A of the type D of discrete type pairs, and for all r_1 and r_2 that are finite sets (over A), there exists a finite set r such that for all x in $|A|$, x belongs to r if and only if x belongs to either r_1 or r_2."

$$\forall A\!:\!D.\, \forall r_1, r_2\!:\!FS(A).\, \exists r\!:\!FS(A).\, \forall x\!:\!|A|.\, x \,\epsilon\{A\}\, r \Leftrightarrow x \,\epsilon\{A\}\, r_1 \vee x \,\epsilon\{A\}\, r_2$$

As usual, a constructive proof of $\exists x.P$ is a pair $<a, p>$ where p proves $P[a/x]$. Thus, as the proof of the above lemma must be constructive, it provides an actual procedure that given a discrete type pair A, and two finite sets r_1 and r_2, returns a pair, where the first component is the union of r_1 and r_2 and the second is constructive evidence of this fact. This procedure is supplied automatically from the completed proof by Nuprl's extractor. An additional library object is created that associates a definition for finite set union (displayed as $\cup\{A\}$) with this extraction. Another typical example is

[3] The term $[nil \rightarrow b; h.t, v \rightarrow w;\; @l]$ is defined as (a hopefully more readable version of) Nuprl's list recursion combinator $list\_ind(l; b; h, t, v.w)$. When l is nil this term reduces to b and when l is $h'.t'$ it reduces to the term w' where w' is w with h' substituted for (free occurrences of) h, t' substituted for t, and $list\_ind(t'; b; h, t, v.w)$ substituted for v.

```
Graph:
A:D # V:FS(A) # E: (elts{A}(V)->elts{A}(V)->U1) #
   ∀x,y:elts{A}(V). E(x,y) ∨ ¬E(x,y) &
   ∀x,y:elts{A}(V). E(x,y) <=> E(y,x) &
   ∀x:elts{A}(V). ¬E(x,x)

n→ (l1,l2):
λ n l1 l2. ∀G:Graph. n ≤ |V(G)| =>
  ∃s:FS(|G|). s ⊂ {|G|} V(G) &
    |s| = l1 & ∀x,y ϵ{|G|} s. ¬(x ={||G||} y) => E(G)(x,y)  ∨
    |s| = l2 & ∀x,y ϵ{|G|} s. ¬E(G)(x,y)

ramsey:
>> ∀l1,l2:{2..}. ∃n:N+. n→ (l1,l2)
```

Figure 1: Graph and Ramsey Theory

the lemma *pick* which states that an element can be picked from any non-empty finite set.

$$\forall A\!:\!D.\, \forall s\!:\!FS(A).\, 0 < |s| \Rightarrow \exists x\!:\!|A|.\, x\,\epsilon\{A\}\, s$$

Several additional definitions leading up to the statement of Ramsey's theorem are provided in Figure 1. The first defines the type of graphs (displayed as *Graph*). A graph is parameterized by a discrete type pair A, and contains a vertex set V, and an edge relation E that is decidable, symmetric, and irreflexive. Not shown are defined projection functions that access the graph's carrier, vertex set, and edge relation components. Their display forms are $|G|$, $V(G)$, and $E(G)$ respectively.[4] The second definition defines a "ramsey function" (displayed as $n \rightarrow (l_1, l_2)$), which states that any graph with at least n vertices contains an l_1-clique or an l_2-independent set. The third object is the statement of Ramsey's theorem itself. $\{2..\}$ and $N+$ represent the set of integers at least two and the positive integers, respectively. In our presentation here, we have used indentation to aid the reader in parsing Nuprl terms. In the actual system, a structure editor is used that has facilities for disambiguating parsing when creating and viewing expressions.

Our development of finite set and graph theory also includes building specialized tactics. Nuprl's standard tactic collection [10] contains a number of modules for automating common forms of logical reasoning. On top of these are special purpose tactics for set theoretic reasoning. The most powerful of these, *FSTactic*, uses a combination of term rewriting, backchaining, propositional reasoning, and congruence reasoning to solve set membership problems. For example, the lemma defining finite set union is automatically used to rewrite membership in a union of sets to a disjunction of membership terms. Our collection consists of about 150 lines of ML code and was written in a day.

4.3 Outline of Main Proof Steps

The actual Nuprl proof of Ramsey's theorem closely follows the informal outline in Section 2, albeit with more detail. The following snapshots represent the main steps and are derived from the actual

[4]Note that we have overloaded display forms (i.e., we have distinct definitions that display the same way) such as vertical bars which project carriers from both graphs and discrete type pairs. Hence, in the second definition in Figure 1, the term $||G||$ is the type given by projecting out the discrete type pair from a graph tuple G and then projecting out the type from the resulting pair.

Nuprl session. These steps convey much of the flavor of proof development in Nuprl and indicate how carefully developed definitions, lemmas, and tactics, facilitate high-level, comprehensible proof development.

(Note that some space saving simplifications have been made. Some hypotheses and uninteresting parts (e.g., applications of *FSTactic*) of refinement steps are omitted. Hypothesis numbers are omitted unless referenced in subsequent snapshots. As the proof references only one discrete type pair, such references are dropped whenever possible. For example, expressions like $s_1 \cup\{A\}\, s_2$ and $x \,\epsilon\{A\}\, s$ are replaced with the simpler $s_1 \cup s_2$ and $x \,\epsilon\, s$. The complete unaltered proof is found in [1].)

The first snapshot is of the initial proof step. At the top is the goal, our statement of Ramsey's theorem. It is followed by a refinement rule, a combination of tactics that specifies induction on l_1 using j_1 as the induction variable. The next two lines contain the hypothesis and subgoal for the base case, and the last three lines contain the inductive case. This simple looking refinement step hides 64 primitive refinement rules. (That is, the tactics in this refinement step internally generate a 64 node proof tree containing only primitive refinement rules. The size of this underlying proof tree, to a first approximation, roughly indicates the degree of automated proof construction.)

```
>> ∀l1,l2:{2..}. ∃n:N+. n→ (l1,l2)
BY OnVar 'l1' (NonNegInd 'j1')
j1 = 2
   >> ∀l2:{2..}. ∃n:N+. n→ (j1,l2)
2 < j1
∀l2:{2..}. ∃n:N+. n→ (j1-1,l2)
   >> ∀l2:{2..}. ∃n:N+. n→ (j1,l2)
```

We continue by refining the base case using the tactic *ITerm* ("Instantiate Term"), which provides the witness l_2 for n. Notice some that abbreviations are expanded by this step, such as $n \to (j_1, l_2)$.

```
BY ITerm 'l2'
G:Graph
l2 ≤ |V(G)|
   >> ∃s:FS(|G|). s ⊂ V(G) &
        |s| = j1 & ∀x,y ϵ s. ¬(x=y in ||G||) => E(G)(x,y) ∨
        |s| = l2 & ∀x,y ϵ s. ¬E(G)(x,y)
```

The result is that, given an arbitrary graph G with at least l_2 vertices, we must find a subset s of its vertex set such that G restricted to s is a 2-clique or an l_2-independent set. To construct such an s, we use a lemma that states that for any decidable predicate P on pairs of elements from some finite set, either P holds for all pairs, or there is a pair for which P fails. This lemma, whose formal statement is

```
>> ∀s:FS(A). ∀P:elts(s)->elts(s)->U1.∀x,y ϵ s.
      P(x)(y) ∨ ¬P(x)(y)
         => ∀x,y ϵ s. P(x)(y) ∨ ∃x,y ϵ s. ¬P(x)(y),
```

is proved by providing (implicitly via an inductive proof) a search procedure that applies P to all x and y in a given s and returns a proof that either P holds for all x and y or returns a pair for which P fails and a proof of $\neg P(x)(y)$. Instantiating P with the appropriate edge relation justifies the following.

```
BY Cases ['∀x,y ε V. (λ z w. z = w in |A| ∨ ¬E(z,w))(x)(y)';
          '∃x,y ε V. ¬((λ z w. z = w in |A| ∨ ¬E(z,w))(x)(y))']
```

When the first case holds, we are provided with a proof of $\neg E(x, y)$, for all x and y in V such that $x \neq y$. As $|V| \geq l_2$, we can pick a subset of V that is an l_2-independent set. In the second case, we are given an x and a y that have an edge between them. Hence, this subset of V is a 2-clique under the edge relation E. It takes 11 refinement steps to complete this analysis.

To prove the inductive case of the first induction, we perform a second induction whose base cases are proved analogously to the first. Afterwards, we have two induction hypotheses, which after several elimination steps are as follows.

```
(6) n→ (j1,j2-1)
(8) m→ (j1-1,j2)
```

These furnish the required Ramsey number for the second induction step.

```
>> ∃n:N+. n→ (j1,j2)
BY ITerm 'n + m'
G:Graph
n+m ≤ |V(G)|
   >> ∃s:FS(|G|). s ⊂ V(G) &
         |s| = j1 & ∀x,y ε s. ¬(x=y in ||G||) => E(G)(x,y) ∨
         |s| = j2 & ∀x,y ε s. ¬E(G)(x,y)
```

After expanding G into its constituent components (a discrete type pair A, a vertex set V, an edge relation E, and edge properties p_1, p_2, and p_3 — i.e., the six parts of the graph tuple defined in Figure 1), we instantiate the outermost quantifiers of the *pick* lemma to select an element v_0 from V. Then, using finite set comprehension, provided by a lemma *fs_comp* which states that any set (here $V - v_0$) may be partitioned by a decidable property (here $\lambda x.\, E(v_0)(x)$), we divide V into r_1 and r_2: those elements of V connected to v_0 and those not.

```
BY InstLemma 'pick' ['A';'V'] THEN InstLemma 'fs_comp' ['A';'V - v0';'E(v0)']
```

This leaves our conclusion unchanged and provides the following new hypotheses.

```
v0 ε V
(19) ∀x:elts(V - v0). x ε r1 <=> x ε V - v0 & E(v0,x) &
                      x ε r2 <=> x ε V - v0 & ¬E(v0,x)
disj(r1,r2)
r1 ∪ r2 = V - v0
```

Using the last two hypotheses and our cardinality lemmas, we prove that $|r_1| + |r_2| \geq m + n - 1$. This takes two refinement steps. It follows (using a tactic for simple monotonicity reasoning) that $|r_1| \geq m \vee |r_2| \geq n$, and we split on the two cases. In the $|r_1| \geq m$ case, we instantiate hypothesis 8, one of our two induction hypotheses as follows.

```
BY EOnThin '<A,<r1,<E,<p1,<p2,p3>>>>>' 8
```

Recall that a Graph is a tuple. The tactic *EOnThin* ("Eliminate On and then Thin (drop) the indicated (uninstantiated) hypothesis") performs an elimination step that instantiates hypothesis 8 with the graph G restricted to the vertex set $r_1 \subset V$. The other components of G are as before. This yields the following hypothesis.

```
(m ≤ |r1|) => ∃s:FS(A). s ⊂ r1 &
   |s| = j1-1 & ∀x,y ∈ s. ¬(x=y in |A|) => E(x,y) ∨
   |s| = j2 & ∀x,y ∈ s.¬E(x,y)
```

Breaking down this hypothesis yields another case split. In the first case, we are given the new hypotheses:

```
(22) s ⊂ r1
|s| = j1-1
(26) ∀x,y ∈ s.¬(x=y in |A|) => E(x,y)
```

Our conclusion remains unchanged. We must still produce a subset s of V that contains a j_1-clique or a j_2-independent set. In this case we prove the left disjunct by introducing the set $s + v_0$ and demonstrate that it constitutes a clique.

```
>> ∃s:FS(A). s ⊂ V &
     |s| = j1 & ∀x,y ∈ s. ¬(x  = y in |A|) => E(x,y) ∨
     |s| = j2 & ∀x,y ∈ s. ¬E(x,y)
BY ITerm 's + v0' THEN ILeft
s + v0 ⊂ V
|s + v0| = j1
x ∈ s + v0
y ∈ s + v0
(33) ¬(x=y in |A|)
   >> E(x,y)
```

This step results in the new goal of proving that G restricted to $s + v_0$ is indeed a clique: that is, given an arbitrary x and y in $s + v_0$, proving $E(x, y)$. The first two hypotheses in the display above in fact correspond to two other subgoals produced by the indicated refinement; however, they are proved automatically by the tactic *FSTactic* mentioned previously.[5] This new goal is proved by examining the four possible cases of $x = v_0$ and $y = v_0$. This type of reasoning is routine and each case takes one refinement step (invoking the suitable tactic or combination of tactics) to verify. If $x = y = v_0$, then this contradicts hypothesis 33. If neither are, than both vertices are in s and $E(x, y)$ follows from hypothesis 26. In the remaining two cases, one vertex is v_0 and the other is not; $E(x, y)$ follows from hypotheses 19 and 22, together with the definition of r_1 and the symmetry of E.

In the second case, our new hypotheses are

```
s ⊂ r1
|s| = j2
∀x,y ∈ s. ¬E(x,y)
```

[5] This does not come for free. *FSTactic* undertakes significant amounts of search and this is slow. Its execution time contributes significantly to our replay time (when the tactic must be rerun to construct the underlying primitive inference tree) and library development time discussed in the next section.

Instantiating the goal with s now proves the existence of a j_2-independent set.

```
>> ∃s:FS(A). s ⊂ V &
     |s| = j1 & ∀x,y ∊ s. ¬(x=y in |A|) => E(x,y) ∨
     |s| = j2 & ∀x,y ∊ s. ¬E(x,y)
BY ITerm 's' THEN IRight
```

FSTactic completes the $|r_1| \geq m$ case. The other case is proved analogously.

Overall, the entire proof consists of 64 refinement steps and took about 20 man-hours to prove, including aborted proof attempts. Tactics played a major role in making this development feasible; the 64 refinement steps hide 17531 primitive steps, an expansion factor of 273 to 1.

4.4 Computational Content

Although our constructive proof may be more complicated than a corresponding classical proof, our proof constructs three interesting functions that can be automatically synthesized by Nuprl's extractor. They are displayed below. Let us note that *term_of*(*thm*) is a term that evaluates to the extraction from a theorem *thm*, i.e., evaluates to a proof object of *thm*. ".1" and ".2" are defined as first and second projection functions on pairs.

ram_n The outermost type constructors of Ramsey's theorem define an *AE* ($\forall/\exists$) formula. Hence, its extraction, $\lambda\, l_1\, l_2.\mathit{term\_of}(\mathit{ramsey})(l_1)(l_2).1$, constitutes a function from integers l_1 and l_2 that evaluates to the first projection of a pair $<n, P_1>$. This function returns n, an upper bound on the Ramsey number for l_1 and l_2.

ram_clique P_1, the second component of the above pair, is the computational part of our proof that $n \to (l_1, l_2)$. This too is an *AE* statement (see Figure 1) and defines a function whose application to a G in *Graph* and a trivial proof (*axiom*) that G is sufficiently large evaluates to $<s, P_2>$. The function $\lambda\, l_1\, l_2\, g.\mathit{term\_of}(\mathit{ramsey})(l_1)(l_2).2(g)(\mathit{axiom}).1$, returns the set s, where G restricted to s is an l_1-clique or an l_2-independent set.

ram_decide P_2 is the computational content of our proof of a disjunction. It provides the basis for a procedure that prints "Clique" or "Independent Set" depending on which disjunct holds.

Nuprl contains facilities to execute these functions. For example, let G be a tuple of type *Graph* that encodes the six vertex "hexagon graph" given in Subsection "Computing" of Section 3. Execution of *ram_n*(3)(3) returns 6, so G must contain three vertices that form a clique or an independent set. Execution of *ram_clique*(3)(3)(G) returns the set 6.(2.(4.nil)) and *ram_decide*(3)(3)(G) prints *Independent Set*. Their execution takes about seven seconds on a Symbolics 3670 Lisp Machine.

5 Comparison

At the heart of the differences between NQTHM and Nuprl are the philosophies underlying the two systems. NQTHM is poised on a delicate balance point between logical strength and theorem proving power. The logic is just strong enough to express many interesting problems, but not so strong that proof automation becomes unmanageable. In the case of Ramsey's theorem, the NQTHM proof was essentially constructive by necessity: as the logic lacks quantifiers, "existence"

	NQTHM	Nuprl
# Tokens	933	972
# Definitions	10	24
# Lemmas	26	25
Replay Time	3.7 minutes	57 minutes

Figure 2: Comparison Statistics

cannot be expressed other than by providing witnessing objects in the theorem statement. Of course, the upside is that around such a restricted logic, Boyer and Moore have been remarkably successful at designing heuristics for automating theorem proving; this is perhaps reflected in the quantitative comparison in Figure 2,[6] and it is certainly reflected in some large proof efforts that have been carried out using that system and PC-NQTHM (see for example [2] and the other articles on system verification in that issue of the Journal of Automated Reasoning). It is also reflected in the total times for the proof efforts. The Nuprl effort took about 60 hours for library development and about 20 additional hours to complete the proof. (However, much of this time was spenting waiting for Nuprl and could be saved by a more efficient version of the system.) The NQTHM effort took about 7 hours altogether, though (as explained in the subsection above entitled *More Statistics and General Remarks*) a few hours may have been saved because of the existence of a previous proof. Finally, the simplicity and classical nature of the NQTHM logic makes it quite accessible to those with little background in logic.

The philosophy behind Nuprl is to provide a foundation for the implementation of constructive mathematics and verified functional programs. Within this framework, the responsibility for automating reasoning falls almost entirely upon the user, though Nuprl does contain a built-in decision procedure for a fragment of arithmetic and Nuprl's standard tactic collection provides significant theorem proving support. Moreover, the logical complexity of Nuprl's type theory is reflected in the complexity of the tactics. For example, term well-formedness is not merely a syntactic property of terms; it is a proof obligation that is undecidable in general, as it is equivalent to showing that a program meets its specification. Nuprl's standard tactic collection contains procedures that in practice solve well-formedness problems (i.e., that a term belongs to some universe), but nonetheless well-formedness is an additional burden on the user and tactic writer and is reflected in development and replay time. However, the richness of the logic contributes to the ease with which problems may be formulated. And its constructivity enables the system to construct interesting programs as results of the theorem-proving process.

It is difficult to compare the naturalness or the ease with which one finds proofs in different systems — especially when the theorem proving paradigms are as different as NQTHM and Nuprl's. In NQTHM, the user incrementally provides definition and lemma statements that lead up to the

[6] In Figure 2, "# Tokens" refers to (essentially) the total number of identifiers and numerals in the input submitted to the systems. It does not count Nuprl tactics or Nuprl definitions or theorem statements leading up to the final proof. Replay times refer to runs on a Sun 3/60 using akcl for the NQTHM run and a Symbolics 3670 Lisp Machine for the Nuprl run. The Nuprl replay time measures the time required to expand (i.e., produce a tree of primitive refinement rules) only the proof of Ramsey's theorem. Nuprl version 3.0 was used for these measurements; a new version is soon to be released that is substantially (up to a factor of 2) more efficient.

desired theorem statement. Each lemma is proved automatically, as is the final theorem. Interaction with the system consists of the user analyzing failed proof attempts and determining what intermediate lemmas, and perhaps hints, are needed to help the prover find a proof. Refinement style proof is possible using the PC-NQTHM interactive enhancement. On the other hand, Nuprl proofs are constructed entirely interactively by refinement.

Interestingly, approximately the same number of lemmas were used for both proofs. However, the lemma statements tend to be rather different. Almost all of the Nuprl lemmas are general purpose propositions about finite sets and are not suggested by failed automated proof attempts. As a result, their statements are often more intuitive than the more technical of the lemmas used in the NQTHM proof. Moreover, the Nuprl proofs seem syntactically close to the style in which mathematics is traditionally presented and provide a formal document of why a theorem is true. Let us return to the question of what is learned from counting the number of lemmas. The sequence of Nuprl refinement steps is probably a more natural analogue of the sequence of NQTHM lemmas than is the sequence of Nuprl lemmas. Both NQTHM lemmas and Nuprl refinement steps are atomic steps carried out automatically without user interaction, and they both take advantage of reasoning capabilities provided by the system (including the tactic libraries, in the case of Nuprl).

Another interesting point, related to proof style, is that, to a certain degree, Nuprl proofs should *not* in general be completely automated. Although any proof is sufficient to demonstrate that a theorem is true, different proofs have different computational content. When algorithmic efficiency is a consideration, the user requires control over the proof steps that affect program complexity. This is not an issue with NQTHM proofs because programs are explicitly given in definitions.

Both Nuprl and NQTHM have strong connections with programming, although the approaches are different. In Nuprl, one may directly verify that an explicitly given program (a term in Nuprl's type theory) meets a specification. However, programs are usually extracted as the implicit "computational content" of proofs. The subsection above entitled *Computational Content* provides examples of the latter approach. In the NQTHM system one must explicitly provide the programs (though they are usually called definitions or "DEFN events") in a language closely related to pure Lisp. However, those programs are translated quite directly by the system into Common Lisp code, which can then be compiled. Therefore these programs can be considerably more efficient than the unoptimized, interpreted programs extracted by Nuprl. (Recall the numbers in the subsections above entitled *Computing* and *Computational Content*: 0.15 seconds to evaluate an NQTHM form (0.011 if the functions are compiled) corresponds to about 7 seconds for evaluation of an analogous Nuprl form, albeit on different machines.) But we can envision improvements in both systems. The Nuprl term language defines a pure functional programming language that is currently interpreted in a straightforward way. It seems reasonable to assume that (with enough effort) execution could be made as efficient as current ML or Lisp implementations. Furthermore, extracted code is currently unoptimized. Work by Sasaki [17] on an optimizer for an earlier version of Nuprl indicates that type information in proofs can be utilized to extract significantly more efficient code. Improvements are also planned for program development in the NQTHM system. Boyer and Moore are developing a successor to NQTHM called `ACL2` that will be based on an applicative subset of Common Lisp. Common Lisp is actually used for serious applications and we expect `ACL2` to be more useful for programmers than the current language.

We have spent little time addressing soundness issues. Both systems are based on well-defined formal logics, so at the very least it is possible to ask whether the systems do indeed implement their respective logics. Both systems have been crafted sufficiently carefully and used sufficiently extensively to give us some confidence that when a statement is certified by the system as being a theorem, then it is indeed a theorem. Another soundness issue is the extent to which a system helps users to develop specifications that reflect their informal intentions. The expressiveness of Nuprl's type theory and the ability to create new notions allow users to express specifications in a reasonably naturally and hierarchical way that follows their logical intuitions. Soundness is not compromised by definitions as the definition facility is essentially a macro facility. Furthermore the "strong typing" in Nuprl prevents certain kinds of specification errors that are analogous to errors prevented by strong typing in programming languages (e.g., array subscripts out of range). The NQTHM logic (hence the prover as well) does allow bona fide definitions, even recursive ones, but guarantees conservativity and hence consistency of the resulting theories. The text printed out by the NQTHM prover is also occasionally useful for discovering errors in specifications. For example, experienced users sometimes detect output indicating obviously false subgoals. Similarly, their suspicions may be raised by proofs that succeed "too quickly".

We have also said little in our comparison about user interface issues. However, in practice, actual usability depends greatly on something less glamorous than the logic or the automated reasoning heuristics, namely the editor. Serious users of NQTHM tend to rely heavily on the capabilities offered by an Emacs editor. Nuprl provides special purpose editors for creating and manipulating definitions, tactics, and proofs.

We should re-emphasize that the numbers presented above, as with many metrics, are potentially misleading. Consider, for example, the problem of comparing sizes of related proofs. We measure this by counting user-entered tokens. But even this simple metric is problematic. Our count measures the number of tokens to be input for the completed proof, and hence ignores the issue of how many tokens were entered on misguided parts of the attempt that never found their way into the final proof. (An attempt to measure this total number of tokens might be informative but might well measure a user's style more than it would measure something about a particular system.) Moreover, even the final token count may be quite dependent on the way one goes about doing the proof, and this can vary wildly among users of a given system. For example, in the Nuprl system, any sequence of proof steps can be encoded as a tactic. In the Nuprl effort, we did not count the tactic code in Figure 2 since the tactics' development was for the most part fairly general. Similarly, arbitrarily large chunks of the proof can be proved separately as lemmas that can be incorporated into the final proof. Nuprl's definition mechanism also allows text to be bound to a single tokens. Comparable difficulties exist in collecting statistics about the NQTHM prover. Many of the tokens come from technical lemmas whose statements were constructed by using the editor to cut and paste; hence the token count is not necessarily reflective of the amount of user interaction. And as with Nuprl, it is easy to shorten the proof with tricks, especially if we use the `let` construct, which is included with the PC-NQTHM interactive enhancement of NQTHM, to share subexpressions. Of the lemmas proved, six (with 65 tokens altogether) were purely about sets, and two were technical lemmas that were created easily using the Emacs editor (without typing many characters, to our recollection) but which accounted for 116 tokens; and all of these were included in the statistics.

In a similar vein, let us point out that a higher definition count might reflect a lack of expressiveness of the logic, or it might instead reflect an elegance of style (modularity). Similarly, the number of lemmas in an interactive system can reflect the user's desire for modularity or it can reflect the weakness of the prover. Replay time can measure the power of the prover, but instead it may measure the extent to which the system "saves" the decisions that were made during the interactive proof of the theorem.

Of course, in our proofs we tried not to exploit such possibilities. Furthermore, to a certain extent, "cheating" in one area is reflected in another. For example a proof may be shortened by increasing definitions and lemmas, and replay time may be decreased by increasing the size (explicitness) of the proof. Hence, quantitative measurements should all be taken together along with the non-quantitative proof aspects in order to aid understanding of the systems in question.

Perhaps the most important point we can make in closing this comparison is that both of us felt comfortable in using the system that we chose, and we each find the other system to be reasonably natural but difficult to imagine using ourselves. It seems to be the case that it takes most people at least a few weeks to get comfortable enough with a proof-checking environment in order to reasonably assess its strengths and weaknesses. We'd like to hope that the descriptions of the proof efforts presented in this paper, together with our comments about the systems, suggest that both systems are quite manageable once one invests some time to become familiar with them. We'd also like to hope that the various warnings presented in this paper encourage people to be cautious about making too-easy judgments regarding the relative merits of systems.

We would be interested in seeing more comparisons of proof development systems. Perhaps though it would be responsible of us to point out that it took much more effort to write this paper than to carry out the proofs.

References

[1] David A. Basin. *Building Theories in Nuprl.* Technical Report 88-932, Cornell University, 1988.

[2] William R. Bevier, Warren A. Hunt, Jr., J Strother Moore, and William D. Young. "An Approach to Systems Verification." Journal of Automated Reasoning, November, 1989.

[3] William R. Bevier. "Kit: A Study in Operating System Verification." IEEE Transactions on Software Engineering, November, 1989, pp. 1368-81.

[4] Robert S. Boyer and J Strother Moore. *A Computational Logic*. Academic Press, New York, 1979.

[5] Robert S. Boyer and J Strother Moore. "Metafunctions: Proving Them Correct and Using Them Efficiently as New Proof Procedures." In *The Correctness Problem in Computer Science*, ed. Robert S. Boyer and J Strother Moore, Academic Press, London, 1981.

[6] Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook.* Academic Press, Boston, 1988.

[7] R.L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, 1986.

[8] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation.* Volume 78 of *Lecture Notes in Computer Science*, Springer-Verlag, 1979.

[9] Ronald L. Graham, Bruce L. Rothschild, and Joel H. Spencer. *Ramsey Theory.* John Wiley and Sons, 1980.

[10] Douglas J. Howe. *Automating Reasoning in an Implementation of Constructive Type Theory.* PhD thesis, Cornell University, 1988.

[11] Matt Kaufmann. *An Example in NQTHM: Ramsey's Theorem.* Internal Note 100, Computational Logic, Inc., November 1988.

[12] Matt Kaufmann. *A User's Manual for an Interactive Enhancement to the Boyer-Moore Theorem Prover.* Technical Report CLI-19, Computational Logic, Inc., May 1988.

[13] Richard A. Kemmerer. *Verification Assessment Study Final Report.* National Computer Security Center, Fort Meade, Maryland, 1986.

[14] Peter A. Lindsay. "A Survey of Mechanical Support for Formal Reasoning." Software Engineering Journal, January, 1988.

[15] Per Martin-Löf. "Constructive mathematics and computer programming." In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, North Holland, Amsterdam, 1982.

[16] J. McCarthy et al. *LISP 1.5 Programmer's Manual.* The MIT Press, Cambridge, Massachusetts, 1965.

[17] James T. Sasaki. *The Extraction and Optimization of Programs from Constructive Proofs.* PhD thesis, Cornell University, 1985.