Building Theories In Nuprl*

David A. Basin
Department of Computer Science,
Cornell University, Ithaca, NY 14853

Abstract

This paper provides an account of how mathematical knowledge is represented, reasoned about, and used computationally in a mechanized constructive theorem proving environment. We accomplish this by presenting a particular theory developed in the Nuprl proof development system: finite set theory culminating in Ramsey's theorem. We believe that this development is interesting as a case study in the relationship between constructive mathematics and computer science. Moreover, the aspects we emphasize — the high-level development of definitions and lemmas, the use of tactics to automate reasoning, and the use of type theory as a programming logic — are not restricted in relevance to this particular theory, and indicate the promise of our approach for other branches of constructive mathematics.

1 Introduction

This paper describes our ongoing research in theory development in the Nuprl proof development system. Nuprl supports the interactive creation of definitions, formulas, proofs, and terms in its formal theory, a sequent calculus formulation of a constructive type theory similar to Martin-Löf's. During the past three years, a number of experiments have been conducted using the system. Howe used Nuprl to prove the fundamental theorem of arithmetic[12] and Girard's paradox[13]. Cleaveland developed the synchronization-tree

*This research was supported in part by NSF grants CCR8502243 and DCR8303327.

model of CCS[3], and Kreitz proved theorems in constructive automata theory[15]. This author has used Nuprl to develop an environment for proving theorems in recursion theory[1] and has recently proved Ramsey's theorem.

Mathematical development in completely formal systems is a relatively new enterprise — starting in 1968 with Automath[7]. A number of systems exist today [2, 6, 16], and the growth of research in this area suggests that such systems will become more important in the future. We believe that Nuprl is particularly well suited for theory development. Its logic is expressive. Special purpose editors encourage users to walk through proofs, expand definitions, and see the effects of different theorem proving strategies. Its high-level meta-language supports the development of *tactics*, programs that partially automate theorem proving. Nuprl is also a programming logic and provides facilities to extract and execute the computational content of proofs; proven correct programs are automatically synthesized from completed proofs. This has the additional pedagogic advantage of sensitizing users to the algorithmic nature of their theories. We explore these topics in sections 2 and 5.

The ability to build theories in Nuprl is based on theoretical research in diverse areas such as programming languages and environments, semantics, automated reasoning, applied logic, and constructive mathematics. Conversely, experience using Nuprl provides more than tangible evidence of theoretical advances. It broadens understanding of these areas and spurs new research. Our work demonstrates the feasibility of automating reasoning about a certain class of objects: those based on proofs of the existence of set theoretic operators. In sections 3 and 4, we present a uniform method of constructing characterization lemmas and sequent rewrite tactics from these objects and develop a powerful decision procedure for automating reasoning about set membership propositions. In addition, we show how to reason about these objects in a way abstract to their underlying type theoretic representations.

Our examples come from our recent work developing finite set and Ramsey theory. The theorem that we prove is the finite version of Ramsey's theorem which states that for all positive integers l_1 and l_2, there is an n such that all graphs with at least n vertices either contain an l_1-clique or an l_2-independent set. We believe that our theory is interesting as a case study in the relationship between constructive mathematics and computer science. Moreover, the aspects we emphasize — the high-level development of definitions and lemmas, the use of tactics to automate reasoning, and the use of type theory as a programming logic — are not restricted in relevance to this particular theory, and indicate the promise of our approach for other branches of constructive mathematics.

2 Nuprl

Requisite to understanding Nuprl theory development is understanding Nuprl itself. Due to space limitations, we offer only a brief overview. The reader is referred to [4] for details.

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

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

To judge a sequent true essentially means that when given members x_i of type H_i, one can construct an inhabitant of A. Nuprl's rules are applied in a top down fashion. That is, they allow us to refine a goal obtaining subgoal sequents such that a goal inhabitant can be computed from subgoal inhabitants. Hence, proofs in Nuprl are downward growing trees where each node has associated with it a sequent and a refinement rule. Children correspond to subgoals that result from refinement rule application.

Refinement rules may be either primitive inference rules or ML programs called tactics. Nuprl tactics are similar to those in LCF[10]: 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. Tactics, then, act as derived inference rules; their correctness is justified by the way the type structure of ML is used. For more on the Nuprl tactic mechanism, see [5], and for a description of the standard tactic collection see [14].

Nuprl's type theory is expressive; its intent is to facilitate the formalization of constructive mathematics. Types are stratified in an unbounded hierarchy of universes. U_1 is the first universe and contains all small types or propositions. As U_1 is itself a type, the closure of it and all types contained in it under the type construction operators yields the next level of the hierarchy, U_2. Propositions are represented via the propositions-as-types correspondence. This correspondence gives a direct translation for the usual quantifiers and logical connectives. A proposition is true if and only if the type associated with it is inhabited. Thus, to prove a proposition P of constructive mathematics, one proves "$>>$ T" for the appropriate type T by applying refinement rules until no unproven subgoals exist. If P is true, a proof will produce an inhabitant of T 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.

3 Representation

Mathematical exposition typically follows a predictable pattern: first basic concepts are defined and notation is introduced, and then lemmas and theorems are developed. Similarly, formalizing representations is the first step in building a Nuprl *library*, a linearly ordered collection of definitions, theorems, tactics, and other objects. Definitions allow us to work directly with familiar logical notation instead of with underlying type-theoretic notation. The actual representation is critical as complex mathematical objects can be defined in various ways; different choices may lead to the development of different theories.

The tradeoff among representation choices is often between generality and efficiency. As definitions become more general, they become usable in the development of other theories. A general finite set theory, for example, supports the development of finite graph or lattice theory. On the other hand, overly general definitions can substantially increase proof complexity. Additional burdens arise in showing that objects meet general criteria. Further, it may be difficult to design proofs with efficient algorithms underlying them. We will return to these points at the end of this section.

The definition facility is essentially a macro facility. For basic logic definitions we use this mechanism directly. For more complex definitions, however, we achieve a kind of abstraction through a level of indirection. Instead of directly equating a display form with a term t, t is extracted out of a theorem that contains type information. We illustrate this two step process with an example. Consider the definition D of discrete pairs, those types with a decidable equality paired with their equality decision procedure. The type these pairs inhabit is specified by quantifying over all small types; this specification is a large U_2 type. Thus, we first prove the typing lemma "$>> U_2$" by explicitly introducing the type

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

In the second step, D is defined as the extraction from this typing theorem. This technique has several advantages. Most importantly, it provides a uniform way of associating a type with each defined object. The standard tactic collection takes advantage of this in proving membership goals, relieving us of the burden of continually retyping commonly referenced objects.

The development of Ramsey's theorem necessitates numerous representation choices. It may be stated about relations, graphs, algebras, and other mathematical structures[11]. As a graph-theoretic statement, it is expressible in terms of two-colorings and monocromatic subgraphs or cliques and independent sets. We choose the latter representation and represent graphs by finite sets and finite set relations. Although the various formulations are in some sense equivalent, they lead to the formalization of very different theories.

Our self-contained library contains 48 definitions. Nineteen of these are related to finite sets, four to graphs, and one is relevant to the statement of Ramsey's theorem. The remainder are part of a general library of basic definitions, such as the direct encodings of the logical connectives of predicate calculus, which are useful for any mathematical theory built in Nuprl.

3.1 Finite Set Theory

Figure 1[1] contains the basic finite set definitions. Definition names are annotated by their display forms and are followed by their respective typing theorem and extraction. D is the previously discussed definition of discrete pairs. *D_type* is a type valued function that projects the discrete type, or carrier, from a discrete pair. For example, if A is the discrete pair

$$< Int, \lambda x, y.int\_eq(x; y; inl(axiom); inr(\lambda x.axiom)) >,$$

then $|A|$ returns *Int*. Finite sets are represented by lists, and *lmember* is a type valued function that takes a discrete pair A, a list l of elements in

[1]Actual text and snapshots from Nuprl sessions are displayed in `typewriter font`. Their appearance here, except for annotations and differences in white space, is identical to their appearance on a terminal during a Nuprl session.

```
D:  D
>> U2
T:U1 # ∀x,y:T. x=y in T ∨ ¬(x=y in T)

D_type:   |<A:D>|
>> D -> U1
λA. A.1

lmember:   <x:A> ϵ{<A:U1>} <l:A list>
>> A:U1 -> l:A list -> x:A -> U1
λ A l x. [ nil → False; h.t,v → x=h in A ∨ v; @ l]

FS:  FS(<A:D>)
>> D -> U1
λA. { l:|A| list | [ nil → True; h.t,v → ¬(h ϵ{|A|} t) & v; @ l] }

fmember:   <x:|A|> ϵ{<A:D>} <l:FS(A)>
>> A:D -> FS(A) -> |A| -> U1
λ A s x. x ϵ{|A|} s
```

Figure 1: Basic Finite Set Definitions

$|A|$, and an x in $|A|$, and returns a type that is inhabited if and only if x is in l. This definition uses a primitive recursive search procedure that in the base case returns *False*, which is defined as *Void* the empty type, and in the inductive case returns an inhabited type if either x equals the head of the list, or, recursively, if x is found in the tail. We also use a search procedure to specify finite sets as functions from discrete pairs D to lists l over $|D|$, where no element in l occurs more than once. Hence, a finite set is a list of non-multiply occurring elements from a type with a decidable equality. We complete the basic definitions by defining finite set membership as list membership.

Appendix 1 presents the complete library listing and contains the definitions of additional finite set theoretic operators, predicates, and objects. These definitions are concise, natural, and are defined in as high-level a way as possible. Figure 2 contains a representative sample and illustrates these points. Increasingly complex operations are defined via more basic ones. For example, subset is defined using set membership, and set equality using subset. If we later change our definition of finite sets, only our definition of finite set membership is effected.

Some definitions are not based on previous definitions and instead are based on extractions from proofs. For example, to define *fs_add*, the repre-

```
fsubset:   <s1:FS(A)> ⊂ {<A:D>} <s2:FS(A)>
>> A:D -> FS(A) -> FS(A) -> U1
λ A s1 s2. ∀x ϵ{A} s1. x ϵ{A} s2

fs_eq:   <s1:FS(A)> ={<A:D>} <s2:FS(A)>
>> A:D -> FS(A) -> FS(A) -> U1
λ A s1 s2. s1 ⊂ {A} s2 & s2 ⊂ {A} s1

members:   elts{<A:D>}(<s:FS(A)>)
>> A:D -> FS(A) -> U1
λ A s. { x:|A| | x ϵ{A} s }

fs_add:   <s:FS(A)> +{<A:D>} <x:|A|>
>> A:D -> FS(A) -> |A| -> FS(A)
λ A s x. fs_add_exists(A,x,s).1
```

Figure 2: Finite Set Operators

sentation of set enlarged by one element, we must possess a procedure that adds elements to sets. We prove the theorem *fs_add_exists*, which states that for any discrete pair A, finite set s_1 on A, and member x of $|A|$, there exists a new set s_2 containing x in addition to the elements of s_1. Formally *fs_add_exists* states

$$\begin{aligned} >> \quad & \forall A : D.\, \forall x : |A|.\, \forall s_1 : FS(A).\, \exists s_2 : FS(A). \forall y : |A|. \\ & y\,\epsilon\{A\}\,s_2 \Leftrightarrow x = y \;in\; |A| \vee y\,\epsilon\{A\}\,s_1. \end{aligned}$$

This theorem's computational content is a simple program that given x and s_1, returns a pair[2]. *fs_add* executes this program and returns the witness s_2. We call such theorems *existence lemmas* and the objects they define *existence objects*.

3.2 Graph And Ramsey Theory

We are now in position to define graphs and state Ramsey's theorem. Implicitly, an undirected graph is a pair: a vertex set and an edge relation. Formally, we must make explicit that a graph is parametrized by a discrete pair A, and an edge relation that is decidable, symmetric, and anti-reflexive. Although we naturally think of graphs as pairs, we often make use of these

[2]Recall that the constructive meaning of the statement $\exists x.P$ is a pair $<a, p>$ where a is a witness such that p is a proof of $P[a/x]$.

```
Graph:   Graph
>> U2
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))

ramfunc:   <n:ℕ>→ (<l1:ℕ>,<l2:ℕ>)
>> ℕ+->{2..}->{2..}->U2
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 3: Graph and Ramsey Theory

explicit properties. We will later see that there is little added burden involved in carrying along these extra properties and that there is an easy way to automatically convert implicit representations, such as adjacency lists, to explicit representations.

Graph, our definition of the type inhabited by graphs, is given in figure 3. Not shown are auxiliary definitions, similar to *D_type*, that provide a means of accessing a graph's carrier, vertex set, and edge relation. Their display forms are $|G|$, $V(G)$, and $E(G)$ respectively. With these in hand, we define *ram_func*, the ramsey number function $n \rightarrow (l_1, l_2)$, which states that any graph with at least n vertices contain an l_1-clique or an l_2-independent set. The last object in figure 3 is the statement of Ramsey's theorem. Here, $\{2..\}$ and $N+$ represent the set of integers at least two and the positive integers respectively.

We conclude this section by returning to the topic of representation generality. Since proving Ramsey's theorem, other researchers at Cornell have built upon this library and developed graph algorithms such as depth first search. During this subsequent development, we have found that for certain applications, some of our definitions are too general and others are not general enough. Many graph algorithms operate by continually modifying a graph's edge set. Under our general definition of an edge relation, these algorithms are inefficient as they cannot assume that the edge relation has a particular structure. On the other hand, if we restrict the edge relation to be a finite set of finite sets, then our definition of discrete pairs is not general enough. Recall that finite sets are parameterized by discrete pairs and their definition states that the equality decision procedure decides if two elements are equal in the carrier's type. If we attempted to define the type inhabited by finite sets of finite sets, then the type of finite sets must be the carrier of some discrete pair. But finite sets are based on lists and list equality is different from finite set equality (*fs_eq*). One remedy is to generalize our definition of discrete pairs by adding an equality equivalence relation. However, this complicates proofs considerably as many operations would have to be demonstrated functional in this relation.

4 Reasoning

4.1 Lemmas

In the previous section, we described our methodology for building representations for mathematical objects. In parallel with this development is the buildup of lemmas that state basic facts about these objects. To some degree, lemma creation, like definition creation, is ad hoc, driven by reoccurring proof obligations and a desire to structure our arguments. However, a properly designed lemma library, in conjunction with definitions for the basic mathematical objects, characterizes these objects in an abstract way. Definitions and lemmas can serve a purpose similar to abstract data types and modules in programming languages: They define basic objects and create an interface that is abstract to the underlying representations. If a library is developed this way, modifications to the underlying representations are only reflected in changes to basic definitions and lemmas.

Many of the finite set theory lemmas are existence lemmas and read like axioms from a set theory text. These lemmas include the existence of the empty set, union, comprehension, and member addition and deletion operations. Their statements are straightforward and are similar to *fs_add_exists*. Figure 4 contains *fs_comp*, which states that a decidable predicate P splits a set into two disjoint sets and P is true for members of the first and is false for members of the second.

As discussed in section 3.1, existence lemmas directly support the definition of basic set theoretic objects. We also use them to abstractly characterize the fundamental properties of these objects. For example, in addition to *fs_add_exists* and its display form, we create a third object, the character-

```
fs_comp:
>> ∀A:D. ∀s:FS(A). ∀P:elts{A}(s)->U1. ∀x:elts{A}(s). P(x) ∨ ¬(P(x))
   => ∃r1,r2:FS(A). ∀x:elts{A}(s). x ϵ{A} r1 <=> x ϵ{A} s & P(x)
      & (x ϵ{A} r2 <=> x ϵ{A} s & ¬(P(x))) & disj{A}(r1,r2) & r1 ∪{A} r2 ={A} s

fs_add_char:
>> ∀A:D. ∀x:|A|. ∀s:FS(A). ∀y:|A|. y ϵ{A} (s +{A} x) <=> x=y in |A| ∨ y ϵ{A} s

fs_induction:
>> ∀A:D. ∀P:FS(A)->Type.
    (P({}) & ∀s:FS(A). ∀x:|A|. ¬(x ϵ{A} s) => (P(s) => P(s +{A} x))) => ∀s:FS(A). P(s)
```

Figure 4: Supporting Lemmas

ization lemma *fs_add_char* found in figure 4; its proof consists of definition expansion. Henceforth, the definition *fs_add* is not expanded; instead, its characterization lemma is used to reason about its properties. This methodology, as the next section demonstrates, also facilitates automated rewriting of set membership propositions.

Not all characterization lemmas are developed in a manner as uniform as those for existence objects. For example, several lemmas are needed to characterize cardinality. These include: adding an element to a set increases its cardinality by one (*fs_add_card*), deleting an element reduces cardinality by one (*fs_del_card*), the cardinality of a set is at least the cardinality of its subsets (*subset_card*), equal sets have equal cardinality (*card_fnl*), and the cardinality of the union of disjoint sets is the sum of their cardinalities (*union_card*).

Other lemmas state reasoning principles such as finite set induction (*fs_induction*, figure 4). Here we take advantage of Nuprl's higher order logic to state a principle holding for all propositions over finite sets. Strictly speaking, induction lemmas for propositions over objects based on integers, lists, or recursive types are unnecessary as we can always expand definitions and perform induction on the underlying type. In fact, that is how these lemmas are proved. However, such lemmas are easy to prove and again provide a reasoning principle abstract to underlying representations.

4.2 Tactics

Tactics, as previously mentioned, are ML programs that partially automate Nuprl theorem proving. They provide a basis for proof construction where the user supplies the main proof ideas and the tactics fill in the details. These high-level steps, combined with comprehensible definitions and lemmas, support the orderly development of readable proofs where the ideas and proof objects are clearly presented.

The design of Nuprl tactic collection and its use is one of the subjects of Howe's thesis[14]. The collection is substantial and includes: components that support common forms of inference, arithmetic and other decision procedures, term and sequent manipulation functions, and many other general purpose programs. The most important tactic is *Autotactic*, which is usually applied after refinement steps to prove resultant subgoals. This tactic succeeds in proving subgoals that follow from certain kinds of arithmetic reasoning and has a component which attempts to prove membership goals, i.e., that some term is a member of some type. Such subgoals arise constantly. For example, demonstrating formula well-formedness takes the form of a membership goal. In general, membership is undecidable as it is equivalent to proving that a program meets a specification. However, *Autotactic* usually succeeds in proving well-formedness. Much of this tactic's success and efficiency results from typing information provided by indirect definitions of user defined objects.

One important feature of the standard collection is that it is written in a general way and provides a large collection of routines that support further tactic development. This makes it easy to write tactics to solve special problems that frequently arise in finite set theory. Most of our tactic development was centered around *FSTactic*, a powerful decision procedure for finite set theory membership problems. An example of such a problem is proving

$$>> x \,\epsilon\{A\}\, (s_2 +\{A\}\, y)$$

under the hypotheses

$$s_1 \subset\{A\}\, s_2 \quad \text{and}$$
$$x \,\epsilon\{A\}\, s_1.$$

These problems frequently occur. For example, the instantiation of any goal $\exists x \,\epsilon\{A\}\, s.\, P$, or the elimination of a hypothesis $\forall x \,\epsilon\{A\}\, s.\, P$, must be followed by a proof that the given witness is actually a member of the appropriate finite set.

FSTactic is based on term rewriting, backchainning, and congruence reasoning. Recall that each existence object comes with a characterization lemma specifying when elements are members of the defined object. For each existence object, we add to *FSTactic* a pair of tactics that automate rewriting via characterization: one tactic rewrites hypotheses and the other conclusions. These tactics attempt to match the right portion of a characterization lemma with a hypothesis or current goal and to replace it with the left portion. Afterwards, appropriate steps are taken to prove the resulting sequent. In the above example, *FSTactic* rewrites the conclusion using *fs_add_char*, generating the new goal

$$>> x \,\epsilon\{A\}\, s_2 \vee x = y \; in \; |A|.$$

After this rewrite, *FSTactic* attempts to prove the right disjunct by congruence reasoning. Failing this, it backchains through set inclusions in an attempt to prove the left disjunct. In our example, this completes the proof. In general, the rewrite process is guaranteed to terminate as each rewrite step reduces the complexity of the finite set operators in the sequent. And when the conclusion follows by membership reasoning, *FSTactic* finds a proof.

Creating special purpose tactics is not difficult. *FSTactic* consists of about 150 lines of ML code and required under 10 hours to write. In practice, it works surprisingly well and routinely proves goals that would require much manual effort.

4.3 Proofs

We have discussed how definitions, lemmas, and tactics serve as the basic building blocks of theories. Now, we examine proofs themselves. Rather than attempting to characterize proof construction, we document our proof of Ramsey's theorem as an example. We shall first present an informal sketch of the proof and then present a semi-formal version: selected snapshots from actual refinement steps.

Our proof, that for all l_1 and l_2 there exists an n such that $n \rightarrow (l_1, l_2)$, proceeds by double induction on l_1 and l_2. In the base case, observe that $l_1 \rightarrow (l_1, 2)$ as any graph with l_1 vertices is either a clique, or there are two vertices that are not connected by an edge. Similarly, $l_2 \to (2, l_2)$. Now, assume by induction that we have some n and m where $n \to (l_1, l_2 - 1)$ and $m \to (l_1 - 1, l_2)$.

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

Proof. Given an arbitrary graph G with 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

$$\begin{array}{rcl} r_1 & = & \{x \in (V - \{v_0\}) : E(v_0, x)\} \text{ and} \\ r_2 & = & \{x \in (V - \{v_0\}) : \neg E(v_0, x)\}. \end{array}$$

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

Our Nuprl proof closely follows the above outline, albeit with more detail. The following snapshots represent the main steps and are taken from the actual Nuprl session.[3] We believe they illustrate our point that careful development of definitions, lemmas, and tactics, facilitates high-level, comprehensible proof development.

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.

[3]Some space saving simplifications have been made: Hypotheses and uninteresting parts of refinement steps are omitted. Hypothesis numbers are omitted unless referenced in subsequent snapshots. As the proof references only one discrete type, such references are dropped whenever possible. For example, expressions like $s_1 \subset\{A\}\, s_2$ and $elts\{A\}(V)$ are replaced with the simpler $s_1 \subset s_2$ and $elts(V)$. Finally, several variables have been renamed to conform to the above proof sketch. The complete unaltered proof is found in appendix 2.

```
>> ∀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*, which provides the witness 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 prove a lemma that states that for any decidable predicate P on (curried) pairs of elements from some finite set, either P hold for all pairs of elements, or there exists 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 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 basis for a case split.

```
BY Cases ['∀x,y ϵ{A} 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 that between all x and y in V, where $x \neq y$, there is an edge. 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 as a hypothesis 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.

```
BY OnVar `l2` (NonNegInd `j2`)
j2 = 2
   >> ∃n:N+. n→(j1,j2)
2 < j2
∃n:N+. n→(j1,j2-1)
   >> ∃n:N+. n→(j1,j2)
```

The proof of the base case of this second induction is analogous to the first base case. Afterwards, we are provided with two inductive 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 inductive 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| = l2 & ∀x,y ϵ s. ¬E(G)(x,y)
```

After expanding G into its constituent components (a discrete pair A, a vertex set V, an edge relation E, and edge properties labeled p_1, p_2, and p_3), we pick an element v_0 from V and divide V into two sets, those with edges to v_0 and those without.

```
BY InstLemma `pick` 'V' THEN InstLemma `fs_comp` ['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
```

From the last two hypotheses along with our cardinality lemmas, we prove that $|r_1| + |r_2| \geq m + n - 1$. This takes two refinement steps. It follows that $|r_1| \geq m \vee |r_2| \geq n$, and we split on the two cases. In the case where $|r_1| \geq m$, we instantiate hypothesis 8, one of our two inductive hypotheses as follows.

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

Recall that a Graph is a 6-tuple. The tactic *EOnThin* 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 ϵ s.
¬(x=y in |A|) => E(x,y) ∨ |s| = j2 & ∀x,y ϵ s.¬E(x,y)
```

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

```
(22) s ⊂ r1
|s| = j1-1
(26) ∀x,y ϵ 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. We are now in a position to prove the left disjunct by introducing the set $s + v_0$ and demonstrating 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 $E(x,y)$. This is accomplished 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 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.

In the second case, our new hypotheses are

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

We demonstrate the existence of a j_2-independent set by instantiating the goal with s

```
>> ∃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 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. As the two base cases are almost identical, as well as the cardinality case split in the inductive case, future analogical reasoning tactics could halve the proof.

```
let ram_n = λ l1 l2. term_of(ramsey)(l1)(l2).1 ;;
let ram_clique = λ l1 l2 g. term_of(ramsey)(l1)(l2).2(g)(axiom).1 ;;
let ram_decide = λ l1 l2 g.
  decide(term_of(ramsey)(l1)(l2).2(g)(axiom).2.2; u."Clique"; u."Independent Set") ;;

let make_graph = λl.
  <<Int, λ x y. int_eq(x;y;inl(axiom);inr(λz.axiom))>,
  <make_list(length(l)),
  <λ x y. lookup(x)(y)(l)(Int)(void),
  <λ x y. lookup(x)(y)(l)(inl(0))(inr(λz.axiom)),
  <λ x y. <λz.0,λz.0>, λx.λz.0>>>>> ;;
```

Figure 5: Ramsey Programs

5 Programming

Nuprl is a system for program synthesis; given a computationally meaningful assertion (e.g., a program specification) and a proof, the extractor can produce a program guaranteed to meet its specification. For example, *fs_add_exists* may be thought of as a program specification. Its extraction is a program that, essentially, adds elements to sets. A number of traditional programs have been developed in this fashion. These include programs that sort, search, and factor numbers.

In constructive mathematics, most interesting theorems have computational content. Ramsey's theorem is no exception. Figure 5 contains three functions that provide access to this content.

ram_n The outermost type constructors of Ramsey's theorem define an *AE* ($\forall/\exists$) formula. Hence, its extraction, accessed with the operator *term_of*, constitutes a function from integers l_1 and l_2 that evaluates to the pair $<n, P_1>$. *ram_n* returns n, an upperbound 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 $n \rightarrow (l_1, l_2)$. This too is an *AE* statement (see *ramfunc*, figure 3) and defines a function whose application to a G in *Graph* and a (trivial) proof that G is sufficiently large evaluates to $<s, P_2>$. *ram_clique* 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. Thus, *ram_decide* provides a decision procedure and prints "Clique" or "Independent Set" depending on which disjunct holds.

In general, programs may be either extracted from proofs or directly coded in Nuprl's object language and stored in library *eval* objects. The programs *ram_n*, *ram_clique*, and *ram_decide* are directly coded but are based on extraction from the theorem *ramsey*. In addition, Figure 5 presents *make_graph*, our program that converts an adjacency list representation of a graph into a tuple of type *Graph*. This program is based on two auxiliary functions *make_list* and *lookup*. The first takes an adjacency list, and returns a finite set, the list of integers between one and the length of the given adjacency list. The second, an adjacency list lookup routine, defines both the edge relation and edge decision procedure. The entire listing is given in appendix 1.

We conclude this section with an example. Let G be the graph isomorphic to a hexagon.

$$\begin{aligned} G &= \mathit{make\_graph}((2.6.nil).(1.3.nil).(2.4.nil). \\ &\quad (3.5.nil).(4.6.nil).(5.1.nil).nil);; \end{aligned}$$

Execution of *ram_n*$(3)(3)$ returns six, so G must contain three vertices that constitute 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 out *Independent Set*. These programs are reasonably efficient, given the intractable nature of the problems they solve. Their execution takes seven seconds on a Symbolics 3670 Lisp Machine.

6 Conclusions And Future Work

We have provided a detailed account of a substantial theory developed in Nuprl. We have emphasized the methodology underlying our development and the computational nature of the resulting theory. The question of how one best understands and implements computational ideas is a difficult one. Our success gives us confidence that the constructive mathematical approach is a major step in the right direction.

Our future research will center on further automating proof construction. The current tactic collection is substantial, but much work remains to be done. Ideally, users should only provide those proof steps requiring creativity, such as the introduction of witnesses or the suggestion of unusual lemmas, and tactics would complete the proof. We have had some success writing such tactics for small domains such as the finite set membership problem whose decision procedure we described in section 4.2. We will work to expand the applicability of such procedures. New techniques such as higher-order unification[8], focus object directed rewriting[16], and rewriting via partial reflection[14] should move us in this direction.

Acknowledgements

by Robert Constable, Doug Howe, Chet Murthy, Mike Quayle, and Scott Smith.

References

[1] David Basin. An environment for automated reasoning about partial functions. In *9th International Conference On Automated Deduction*, pages 101–110, Argonne, Illinois, 1988. Springer-Verlag.

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

[3] W.R. Cleaveland II. *Type-Theoretic Models of Concurrency*. PhD thesis, Cornell, 1987.

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

[5] Robert L. Constable, Todd B. Knoblock, and Joseph L. Bates. Writing programs that construct proofs. *Journal of Automated Reasoning*, 1(3):285–326, 1985.

[6] Thierry Coquand and Gérard Huet. Constructions: A higher order proof system for mechanizing mathematics. In B. Buchberger, editor,

EUROCAL '85: European Conference on Computer Algebra, pages 151–184. Springer-Verlag, 1985.

[7] N.G. deBruijn. A survey of the project automath. In *Essays in Combinatory Logic, Lambda Calculus, and Formalism.*, pages 589–606. Academic Press, 1980.

[8] Amy Felty and Dale Miller. Specifying theorem provers in a higher-order logic programming language. In *9th International Conference On Automated Deduction*, Argonne, Illinois, 1988.

[9] A. Ferro et al. Decision procedures for elementary fragments of set theory. In *Fifth Conference on Automated Deduction*, pages 88–96, Les Arcs, France, 1980.

[10] 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.

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

[12] Douglas J. Howe. Implementing number theory: An experiment with Nuprl. In *8th International Conference On Automated Deduction*, Oxford, UK, 1986.

[13] Douglas J. Howe. The computational behaviour of Girard's paradox. In *Symposium on Logic in Computer Science*, pages 205–214. Computer Society Press of the IEEE, 1987.

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

[15] Christoph Kreitz. Constructive automata theory implemented with the Nuprl proof development system. Technical Report 86-779, Cornell University, 1986.

[16] David A. McAllester. *ONTIC: A Knowledge Representation System for Mathematics*. MIT Press, 1989.

[17] Lawrence C. Paulson. Lessons learned from LCF: a survey of natural deduction proofs. *Comp. J.*, 28(5), 1985.

Appendix 1 — Finite Set Library

```
* DEF fs
  (<T:tactic>!)==
  (\p.((<T>) THEN Repeat (FSTactic ORELSE WeakAutotactic)) p)

* THM D_
  >> U2
  Extraction:
  T:U1 # ∀x,y:T. x=y in T ∨ ¬(x=y in T)

* DEF D
  D==D

* THM D_type_
  >> D -> U1
  Extraction:
  λA. A.1

* DEF D_type
  |<A:D>|==D_type(<A>)

* THM lmember_
  >> A:U1 -> l:A list -> x:A -> U1
  Extraction:
  λ A l x. [ nil → False; h.t,v → v ∨ x=h in A; @ l]

* DEF lmember
  <x:A> ϵ{<A:U1>} <l:A list>==lmember(<A>,<l>,<x>)

* THM FS_
  >> D -> U1
  Extraction:
  λA. { l:|A| list | [ nil → True; h.t,v → v & ¬(h ϵ{|A|} t); @ l] }

* DEF FS
  FS(<A:D>)==FS(<A>)

* THM fmember_
  >> A:D -> FS(A) -> |A| -> U1
  Extraction:
  λ A s x. x ϵ{|A|} s

* DEF fmember
  <x:|A|> ϵ{<A:D>} <l:FS(A)>==fmember(<A>,<l>,<x>)

* DEF all_fmembers
  ∀<x:var> ϵ{<A:D>} <s:FS(A)>. <P:prop>==
  ∀<x>:|<A>|. <x> ϵ{<A>} <s> => <P>

* DEF all2_fmembers
  ∀<x:var>,<y:var> ϵ{<A:D>} <s:FS(A)>. <P:prop>==∀<x>,<y>:|<A>|. <x> ϵ{<A>} <s> & <y> ϵ{<A>} <s>
=> <P>

* DEF some_fmember
```

```
  ∃<x:var> ϵ{<A:D>} <s:FS(A)>. <P:prop>==
  ∃<x>:|<A>|. <x> ϵ{<A>} <s> & <P>

* DEF some2_fmember
  ∃<x:var>,<y:var> ϵ{<A:D>} <s:FS(A)>. <P:prop>==∃<x> ϵ{<A>} <s>. ∃<y> ϵ{<A>} <s>. <P>

* ML FSMember
  letref defs_to_unfold_for_FSMember = ``fs_eq fsubset``;;

  let FSMember =
    FastAp (Repeat (Unfolds defs_to_unfold_for_FSMember)
            THEN Backchain);;

  add_to_autotactic `FSMember`
  (IfOnConcl (\c. member (ext_name c)
                    (`fmember` . defs_to_unfold_for_FSMember))
             FSMember
             Fail) ;;

* THM fsubset_
  >> A:D -> FS(A) -> FS(A) -> U1
  Extraction:
  λ A s1 s2. ∀x ϵ{A} s1. x ϵ{A} s2

* DEF fsubset
  <s1:FS(A)> ⊂ {<A:D>} <s2:FS(A)>==fsubset(<A>,<x>,<l>)

* THM fs_eq_
  >> A:D -> FS(A) -> FS(A) -> U1
  Extraction:
  λ A s1 s2. s1 ⊂ {A} s2 & s2 ⊂ {A} s1

* DEF fs_eq
  <s1:FS(A)> ={<A:D>} <s2:FS(A)>==fs_eq(<A>,<s1>,<s2>)

* THM fs_null_
  >> Object
  Extraction:
  nil

* DEF fs_null
  {}==fs_null

* THM fs_null__
  >> ∀A:D. {} in FS(A)

* ML add_fs_null
  add_to_member_i `fs_null`
  (IfOnConcl λc. first_equand c = '{}'
    (Lemma `fs_null__`)
    Fail) ;;

* THM D_type_eq_decidable
  >> ∀A:D. ∀x,y:|A|. x=y in |A| ∨ ¬(x=y in |A|)
```

```
* ML add_D_type_eq_decidable
  add_to_Decidable `D_type_eq_decidable`
  (Lemma `D_type_eq_decidable`);;

* THM fmember_dec
  >> ∀A:D. ∀x:|A|. ∀s:FS(A). x ε{A} s ∨ ¬(x ε{A} s)

* ML add_fmember_dec
  add_to_Decidable `fmember_dec` (Lemma `fmember_dec`);;

* THM fs_add_exists
  >> ∀A:D. ∀x:|A|. ∀s1:FS(A). ∃s2:FS(A). ∀y:|A|.
     y ε{A} s2 <=> x=y in |A| ∨ y ε{A} s1

* THM fs_add_
  >> A:D -> FS(A) -> |A| -> FS(A)
  Extraction:
  λ A s x. fs_add_exists(A,x,s).1

* DEF fs_add
  <s:FS(A)> +{<A:D>} <x:|A|>==fs_add(<A>,<s>,<x>)

* THM fs_add_char
  >> ∀A:D. ∀x:|A|. ∀s:FS(A). ∀y:|A|.
      y ε{A} (s +{A} x) <=> x=y in |A| ∨ y ε{A} s

* THM card_
  >> Object
  Extraction:
  λs. [ nil → 0; h.t,v → 1+v; @ s]

* DEF card
  |<s:FS(A)>|==card(<s>)

* THM card__
  >> ∀A:D. ∀s:FS(A). |s| in N

* THM fs_induction
  >> ∀A:D. ∀P:FS(A)->Type.
     (P({}) & ∀s:FS(A). ∀x:|A|. ¬(x ε{A} s) => P(s) => P(s +{A} x))
     => ∀s:FS(A). P(s)

* ML FSInduction
  let FSInduction t =
    AbstractConcl t THEN Lemma `fs_induction`
    THENM ReduceConcl
  ;;

* THM fs_unroll
  >> ∀A:D. ∀P:FS(A)->U1.
     P({}) => (∀x:|A|. ∀s:FS(A). ¬(x ε{A} s) => P(s +{A} x)) =>
     ∀s:FS(A). P(s)

* ML FSUnroll
  let FSUnroll t =
```

```
    AbstractConcl t THEN Lemma ‘fs_unroll‘
    THENM ReduceConcl ;;

* THM fs_add_card
  >> ∀A:D. ∀s:FS(A). ∀x:|A|. ¬(x ϵ{A} s) => |s +{A} x| = |s|+1 in Int

* THM fs_del_exists
  >> ∀A:D. ∀s1:FS(A). ∀x:|A|. ∃s2:FS(A).
       (∀y:|A|. y ϵ{A} s2 <=> y ϵ{A} s1 & ¬(y=x in |A|))
     & (x ϵ{A} s1 => |s2| = |s1|-1)

* THM fs_del_
  >> A:D -> FS(A) -> |A| -> FS(A)
  Extraction:
  λ A s x. fs_del_exists(A,s,x).1

* DEF fs_del
  <s:FS(A)> -{<A:D>} <x:|A|>==fs_del(<A>,<s>,<x>)

* THM fs_del_char
  >> ∀A:D. ∀x:|A|. ∀s:FS(A). ∀y:|A|. y ϵ{A} (s -{A} x) <=> ¬(x=y in |A|) & y ϵ{A} s

* THM fs_del_card
  >> ∀A:D. ∀s:FS(A). ∀x:|A|. x ϵ{A} s => |s -{A} x| = |s|-1 in Int

* THM subset_card
  >>∀A:D. ∀s1,s2:FS(A). s1 ⊂ {A} s2 => |s1|≤ |s2|

* THM card_fnl
  >> ∀A:D. ∀s1,s2:FS(A). s1 ={A} s2 => |s1| = |s2|

* THM fs_union_exists
  >> ∀A:D. ∀r1,r2:FS(A). ∃r:FS(A). ∀x:|A|.
      x ϵ{A} r <=> x ϵ{A} r1 ∨ x ϵ{A} r2

* THM fs_union_
  >> A:D -> FS(A) -> FS(A) -> FS(A)
  Extraction:
  λ A s1 s2. fs_union_exists(A,s1,s2).1

* DEF fs_union
  <s1:FS(A)> ∪{<A:D>} <s2:FS(A)>==fs_union(<A>,<s1>,<s2>)

* THM fs_union_char
  >> ∀A:D. ∀s1,s2:FS(A). ∀x:|A|.
     x ϵ{A} s1 ∪{A} s2 <=> x ϵ{A} s1 ∨ x ϵ{A} s2

* THM eq_union_char
  >>∀A:D. ∀r1,r2,s:FS(A). r1 ∪{A} r2 ={A} s => ∀x:|A|. x ϵ{A} r1 ∨ x ϵ{A} r2 => x ϵ{A} s

* THM disjoint_
  >> A:D -> FS(A) -> FS(A) -> U1
  Extraction:
  λ A s1 s2. ¬(∃x:|A|. x ϵ{A} s1 & x ϵ{A} s2)
```

```
* DEF disjoint
  disj{<A:D>}(<s1:FS(A)>,<s2:FS(A)>)==disjoint(<A>,<s1>,<s2>)

* THM plus_union_commute
  >>∀A:D. ∀s1,s2:FS(A). ∀x:|A|. s1 +{A} x ∪{A} s2 ={A} s1 ∪{A} s2 +{A} x

* THM union_card
  >> ∀A:D. ∀s1,s2:FS(A). disj{A}(s1,s2) => |s1 ∪{A} s2| = |s1| + |s2|

* THM pick
  >> ∀A:D. ∀s:FS(A). 0<|s| => ∃x:|A|. x ε{A} s

* THM small_card_subset
  >>∀A:D. ∀n:ℕ. ∀s:FS(A). n≤ |s| => ∃r:FS(A). r ⊂ {A} s & |r| = n

* THM members_
  >> A:D -> FS(A) -> U1
  Extraction:
  λ A s. { x:|A| | x ε{A} s }

* DEF members
  elts{<A:D>}(<s:FS(A)>)==members(<A>,<s>)

* THM doubleton_
  >>A:D->|A|->|A|->FS(A)
  Extraction:
  λ A x y. {} +{A} x +{A} y

* DEF doubleton
  {<x:|A|>,<y:|A|>:<A:D>}==doubleton(<A>,<x>,<y>)

* THM doubleton_char
  >>∀A:D. ∀x,y,z:|A|. z ε{A} {x,y:A} <=> z = x in |A| ∨ z = y in |A|

* THM doubleton_card
  >>∀A:D. ∀x,y:|A|. ¬(x = y in |A|) => |{x,y:A}| = 2

* DEF elt_eq
  <x:elt> ={<S:carrier>} <y:elt>==(<x>=<y> in <S>)

* THM fs_comp
  >>∀A:D. ∀s:FS(A). ∀P:elts{A}(s)->U1. ∀x:elts{A}(s). P(x) ∨ ¬(P(x)) => ∃r1,r2:FS(A). ∀x:elts{A}(s).
x ε{A} r1 <=> x ε{A} s & P(x) & x ε{A} r2 <=> x ε{A} s & ¬(P(x)) & disj{A}(r1,r2) & r1 ∪{A} r2
={A} s

* THM Graph_
  >> U2
  Extraction:
  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))

* DEF Graph
  Graph==Graph

* DEF Graph_carrier
```

```
  |<G:Graph>|==Graph_carrier(<G>)

* THM Graph_carrier_
  >>G:Graph->D
  Extraction:
  λG. G.1

* THM vertex_
  >>G:Graph->FS(|G|)
  Extraction:
  λG. G.2.1

* DEF vertex
  V(<G:Graph>)==vertex(G)

* THM edges_
  >>G:Graph->elts{|G|}(V(G))->elts{|G|}(V(G))->U1
  Extraction:
  λG. G.2.2.1

* DEF edges
  E(<G:graph>)==edges(<G>)

* THM ramfunc_
  >>N+->{2..}->{2..}->U2
  Extraction:
  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))

* DEF ramfunc
  <n:N>→ (<l1:N>,<l2:N>)==ramfunc(<n>)(<l1>)(<l2>)

* ML GraphE
  let GraphE newids i =
    GenProductE newids i THEN
    UnfoldsInConcl ``Graph_carrier vertex edges`` THEN ReduceConcl THEN Thin i ;;

* THM fset_prop
  >>∀A:D. ∀s:FS(A). ∀P:elts{A}(s)->U1. ∀x ϵ{A} s. P(x) ∨ ¬(P(x)) => ∀x ϵ{A} s. P(x) ∨ ∃x ϵ{A}
s. ¬(P(x))

* THM fset2_prop
  >>∀A:D. ∀s:FS(A). ∀P:elts{A}(s)->elts{A}(s)->U1. ∀x,y ϵ{A} s. P(x)(y) ∨ ¬(P(x)(y)) => ∀x,y
ϵ{A} s. P(x)(y) ∨ ∃x,y ϵ{A} s. ¬(P(x)(y))

* ML DecideCases
  let MakeDecideCases v1 v2 n p =
  ( Decide (make_equal_term (get_type p v1) [v1;v2]) THEN
    InstHyp [v1;v2] n THENM OnLastHyp E
  ) p ;;

  let SolveDecideCases = %Decide Trivial Cases %
   Complete (ILeft THENM ILeft...) ORELSE
   Complete (ILeft THENM IRight ...) ORELSE
   Complete (IRight THENM ILeft...) ORELSE
   Complete (IRight THEN IRight...) ORELSE
```

```
  % Double Negation Case %
  Complete (IRight THENM I THENM OnLastHyp E THENM
             Backchain...) ;;

 let DecideCases v1 v2 n =
   MakeDecideCases v1 v2 n THENM Try SolveDecideCases ;;

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


* EVAL demo
 % list procedures %
 let hd = λl. [ nil →  "?"; h.t,v →  h; @ l] ;;
 let tl = λl. [ nil →  nil; h.t,v →  t; @ l] ;;

 let nthtl = λ n l.
   λx. [ 0 →  x ; i, v →  tl(v) ; @ n](l)
  ;;

 let member = λ x l tt ff. % i x in l return tt else ff %
   [ nil →  ff; h.t,v →  if x=h then tt else v; @ l] ;;

 let lookup = λ x y G tt ff. % lookup in adjacency list %
   λl. member(y)(l)(tt)(ff) (hd (nthtl(x-1,G))) ;;

 let make_list = λn. [ 0 →  (nil) ; i, v →  (i.v) ; @ n] ;;
 let length = λl. [ nil →  0; h.t,v →  1+v; @ l] ;;

 % Graph Making Procedures. Make an integer graph from
    an adjacency list.  Recall a graph is a 4-tuple:
       Discrete Type # Vertex Set # Edge Relation #
       Edge Decision Procdure & Other Edge Properties %

 let make_graph = λl.
   <<int, λ x y. int_eq(x;y;inl(axiom);inr(λz.axiom))>,
   <make_list(length(l)),
   <λ x y. lookup(x)(y)(l)(int)(void),
   <λ x y. lookup(x)(y)(l)(inl(0))(inr(λz.axiom)),
   <λ x y. <λz.0,λz.0>, λx . λz.0>>>>> ;;

 let ram = term_of(ramsey) ;;
 let ram_n = λ l1 l2. ram(l1)(l2).1 ;;
 let ram_clique = λ l1 l2 g. ram(l1)(l2).2(g)(axiom).1 ;;
 let ram_decide = λ l1 l2 g.
   decide(ram(l1)(l2).2(g)(axiom).2.2; u."Clique";
          u."Independent Set") ;;
```

Appendix 2 — Ramsey's Theorem

```
>> ∀l1,l2:{2..}. ∃n:N+. n→ (l1,l2)
| BY (OnVar `l1` (NonNegInd `j1`)...*)
| 1. j1:int
| 2. j1 = 2
|- >> ∀l2:{2..}. ∃n:N+. n→ (j1,l2)
| | BY (I THENM ITerm 'l2'...)
| | 3. l2:{2..}
| | 4. G:Graph
| | 5. l2≤ |V(G)|
| |- >> ∃s:FS(|G|). s ⊂ {|G|} V(G) & |s| = j1 & ∀x,y ϵ{|G|} s. ((¬(x=y in ||G||))->E(G)(x,y))
∨ |s| = l2 & ∀x,y ϵ{|G|} s. (¬(E(G)(x,y)))
| | | BY (BringHyp 5 THEN GraphE ``A V E`` 4...)
| | | 4. A:D
| | | 5. V:FS(A)
| | | 6. E:elts{A}(V)->elts{A}(V)->U1
| | | 7. x5:∀x,y:elts{A}(V). E(x,y) ∨ ¬(E(x,y))
| | | 8. x7:∀x,y:elts{A}(V). E(x,y) <=> E(y,x)
| | | 9. x8:∀x:elts{A}(V). ¬(E(x,x))
| | | 10. l2≤ |V|
| | |- >> ∃s:FS(A). s ⊂ {A} V & |s| = j1 & ∀x,y ϵ{A} s. ((¬(x=y in |A|))->E(x,y)) ∨ |s| = l2
& ∀x,y ϵ{A} s. (¬(E(x,y)))
| | | | BY (Cases ['∀x,y ϵ{A} V. λ z w. z = w in |A| ∨ ¬(E(z,w))(x)(y)';'∃x,y ϵ{A} V. ¬(λ z
w. z = w in |A| ∨ ¬(E(z,w))(x)(y))']...)
| | | |- >> ∀x,y ϵ{A} V. λ z w. z=w in |A| ∨ ¬(E(z,w))(x,y) ∨ ∃x,y ϵ{A} V. ¬(λ z w. z=w in |A|
∨ ¬(E(z,w))(x,y))
| | | | | BY (Lemma `fset2_prop` THEN ReduceConcl...) THEN (DecideCases 'x' 'y' 7...)
| | | | 11. ∀x,y ϵ{A} V. λ z w. z=w in |A| ∨ ¬(E(z,w))(x,y)
| | | |- >> ∃s:FS(A). s ⊂ {A} V & |s| = j1 & ∀x,y ϵ{A} s. ((¬(x=y in |A|))->E(x,y)) ∨ |s| =
l2 & ∀x,y ϵ{A} s. (¬(E(x,y)))
| | | | | BY (InstLemma `small_card_subset` ['A';'l2';'V']...) THEN OnLastHyp EThin
| | | | | 12. a14:FS(A)
| | | | | 13. a14 ⊂ {A} V & |a14| = l2
| | | | |- >> ∃s:FS(A). s ⊂ {A} V & |s| = j1 & ∀x,y ϵ{A} s. ((¬(x=y in |A|))->E(x,y)) ∨ |s|
= l2 & ∀x,y ϵ{A} s. (¬(E(x,y)))
| | | | | | BY (ITerm 'a14'...) THEN (IRight...)
| | | | | | 13. a14 ⊂ {A} V
| | | | | | 14. |a14| = l2
| | | | | | 15. a14 ⊂ {A} V
| | | | | | 16. |a14| = l2
| | | | | | 17. x:|A|
| | | | | | 18. y:|A|
| | | | | | 19. x ϵ{A} a14
| | | | | | 20. y ϵ{A} a14
| | | | | | 21. E(x,y)
| | | | | |- >> void
| | | | | | | BY (InstHyp ['x';'y'] 11 THENS OnLastHyp E THEN Try Backchain...)
| | | | | | | 22. x=y in |A| ∨ ¬(E(x,y))
| | | | | | | 23. x=y in |A|
| | | | | | |- >> void
| | | | | | | | BY (SubstForInHyp 'y=x in elts{A}(V)' 21!)
| | | | | | | |    THEN OnLastHyp λi. BringHyp i THEN (Backchain...)
| | | | 11. ∃x,y ϵ{A} V. ¬(λ z w. z=w in |A| ∨ ¬(E(z,w))(x,y))
```

```
| | | |- >> ∃s:FS(A). s ⊂ {A} V & |s| = j1 & ∀x,y ϵ{A} s. ((¬(x=y in |A|))->E(x,y)) ∨ |s| =
12 & ∀x,y ϵ{A} s. (¬(E(x,y)))
| | | | | BY (RepeatProductE ``x y`` 11 THENM ITerm '{x,y:A}'...) THEN
| | | | |    ((OnLastHyp λi. HypChar `doubleton_char` i THENS OnLastHyp E !!)
| | | | |       ORELSE (ILeft...))
| | | | | 11. x:|A|
| | | | | 12. x ϵ{A} V
| | | | | 13. y:|A|
| | | | | 14. y ϵ{A} V
| | | | | 15. ¬(λ z w. z=w in |A| ∨ ¬(E(z,w))(x,y))
| | | | | 16. {x,y:A} ⊂ {A} V
| | | | |- >> |{x,y:A}| = j1
| | | | | | BY (SubstFor (h 2 p)...) THEN (Lemma `doubleton_card`...)
| | | | | |    THEN (E 15 THENM ReduceConcl...) THEN (ILeft...)
| | | | | 11. x:|A|
| | | | | 12. x ϵ{A} V
| | | | | 13. y:|A|
| | | | | 14. y ϵ{A} V
| | | | | 15. ¬(λ z w. z=w in |A| ∨ ¬(E(z,w))(x,y))
| | | | | 16. {x,y:A} ⊂ {A} V
| | | | | 17. |{x,y:A}| = j1
| | | | | 18. x@0:|A|
| | | | | 19. y@0:|A|
| | | | | 20. x@0 ϵ{A} {x,y:A}
| | | | | 21. y@0 ϵ{A} {x,y:A}
| | | | | 22. ¬(x@0=y@0 in |A|)
| | | | |- >> E(x@0,y@0)
| | | | | | BY (Assert 'E(x,y)' THENM
| | | | | |    λf. f 21 THEN f 20 (λi. HypChar `doubleton_char` i)...)
| | | | | |    THEN Try (Complete (OnLastHyp E THEN UnfoldInConcl `doubleton` !!))
| | | | | | 20. ¬(x@0=y@0 in |A|)
| | | | | | 21. y@0=x in |A| ∨ y@0=y in |A|
| | | | | | 22. x@0=x in |A| ∨ x@0=y in |A|
| | | | | |- >> E(x,y)
| | | | | | | BY (Decide 'x=y in |A|' THEN InstHyp ['x';'y'] 7 THENM OnLastHyp E...)
| | | | | | |    THEN (E 15 THEN ReduceConcl...) THEN (IRight...)
| | | | | | 20. ¬(x@0=y@0 in |A|)
| | | | | | 21. E(x,y)
| | | | | | 22. y@0=x in |A| ∨ y@0=y in |A|
| | | | | | 23. x@0=x in |A| ∨ x@0=y in |A|
| | | | | |- >> E(x@0,y@0)
| | | | | | | BY (E 22 THEN E 23...) THEN Try (Complete (E 20...))
| | | | | | | 24. y@0=x in |A|
| | | | | | | 25. x@0=y in |A|
| | | | | | |- >> E(x@0,y@0)
| | | | | | | | BY (SubstForInHyp 'x=y@0 in elts{A}(V)' 21 !) THEN
| | | | | | | |    (OnLastHyp (SubstForInHyp 'y=x@0 in elts{A}(V)') !!)
| | | | | | | |    THEN (BackThruHyp 8!)
| | | | | | | 24. y@0=y in |A|
| | | | | | | 25. x@0=x in |A|
| | | | | | |- >> E(x@0,y@0)
| | | | | | | | BY (SubstForInHyp 'x=x@0 in elts{A}(V)' 21 !) THEN
| | | | | | | |    (OnLastHyp (SubstForInHyp 'y=y@0 in elts{A}(V)') !!)
| 1. j1:int
| 2. 2<j1
```

```
| 3. ∀l2:{2..}. ∃n:N+. n→ (j1-1,l2)
|- >> ∀l2:{2..}. ∃n:N+. n→ (j1,l2)
| | BY (OnVar ‘l2‘ (NonNegInd ‘j2‘)...)
| | 4. j2:int
| | 5. j2 = 2
| |- >> ∃n:N+. n→ (j1,j2)
| | | BY (ITerm ’j1’...)
| | | 6. G:Graph
| | | 7. j1≤ |V(G)|
| | |- >> ∃s:FS(|G|). s ⊂ {|G|} V(G) & |s| = j1 & ∀x,y ϵ{|G|} s. ((¬(x=y in ||G||))->E(G)(x,y))
∨ |s| = j2 & ∀x,y ϵ{|G|} s. (¬(E(G)(x,y)))
| | | | BY (BringHyp 7 THEN GraphE ‘‘A V E‘‘ 6...)
| | | | 6. A:D
| | | | 7. V:FS(A)
| | | | 8. E:elts{A}(V)->elts{A}(V)->U1
| | | | 9. x6:∀x,y:elts{A}(V). E(x,y) ∨ ¬(E(x,y))
| | | | 10. x8:∀x,y:elts{A}(V). E(x,y) <=> E(y,x)
| | | | 11. x9:∀x:elts{A}(V). ¬(E(x,x))
| | | | 12. j1≤ |V|
| | | |- >> ∃s:FS(A). s ⊂ {A} V & |s| = j1 & ∀x,y ϵ{A} s. ((¬(x=y in |A|))->E(x,y)) ∨ |s| =
j2 & ∀x,y ϵ{A} s. (¬(E(x,y)))
| | | | | BY (Cases [’∀x,y ϵ{A} V. λ z w. z = w in |A| ∨ E(z,w)(x)(y)’;’∃x,y ϵ{A} V. ¬(λ z w.
z = w in |A| ∨ E(z,w)(x)(y))’]...)
| | | | |- >> ∀x,y ϵ{A} V. λ z w. z=w in |A| ∨ E(z,w)(x,y) ∨ ∃x,y ϵ{A} V. ¬(λ z w. z=w in |A|
∨ E(z,w)(x,y))
| | | | | | BY (Lemma ‘fset2_prop‘ THEN ReduceConcl...) THEN (DecideCases ’x’ ’y’ 9...)
| | | | | 13. ∀x,y ϵ{A} V. λ z w. z=w in |A| ∨ E(z,w)(x,y)
| | | | |- >> ∃s:FS(A). s ⊂ {A} V & |s| = j1 & ∀x,y ϵ{A} s. ((¬(x=y in |A|))->E(x,y)) ∨ |s|
= j2 & ∀x,y ϵ{A} s. (¬(E(x,y)))
| | | | | | BY (InstLemma ‘small_card_subset‘ [’A’;’j1’;’V’]...) THEN (OnLastHyp EThin...)
| | | | | | 14. a13:FS(A)
| | | | | | 15. a13 ⊂ {A} V
| | | | | | 16. |a13| = j1
| | | | | |- >> ∃s:FS(A). s ⊂ {A} V & |s| = j1 & ∀x,y ϵ{A} s. ((¬(x=y in |A|))->E(x,y)) ∨ |s|
= j2 & ∀x,y ϵ{A} s. (¬(E(x,y)))
| | | | | | | BY (ITerm ’a13’ ...) THEN (ILeft...)
| | | | | | | 17. a13 ⊂ {A} V
| | | | | | | 18. |a13| = j1
| | | | | | | 19. x:|A|
| | | | | | | 20. y:|A|
| | | | | | | 21. x ϵ{A} a13
| | | | | | | 22. y ϵ{A} a13
| | | | | | | 23. ¬(x=y in |A|)
| | | | | | |- >> E(x,y)
| | | | | | | | BY (InstHyp [’x’;’y’] 13 THENS OnLastHyp E THEN Try Backchain...)
| | | | | | | | 24. x=y in |A| ∨ E(x,y)
| | | | | | | | 25. x=y in |A|
| | | | | | | |- >> E(x,y)
| | | | | | | | | BY Contradiction
| | | | | 13. ∃x,y ϵ{A} V. ¬(λ z w. z=w in |A| ∨ E(z,w)(x,y))
| | | | |- >> ∃s:FS(A). s ⊂ {A} V & |s| = j1 & ∀x,y ϵ{A} s. ((¬(x=y in |A|))->E(x,y)) ∨ |s|
= j2 & ∀x,y ϵ{A} s. (¬(E(x,y)))
| | | | | | BY (RepeatProductE ‘‘x y‘‘ 13 THENM ITerm ’{x,y:A}’...)
| | | | | |    THEN ((OnLastHyp (HypChar ‘doubleton_char‘) THENS OnLastHyp E!)
| | | | | |         ORELSE (IRight...))
| | | | | | 13. x:|A|
```

```
| | | | | | 14. x ∈{A} V
| | | | | | 15. y:|A|
| | | | | | 16. y ∈{A} V
| | | | | | 17. ¬(λ z w. z=w in |A| ∨ E(z,w)(x,y))
| | | | | | 18. {x,y:A} ⊂ {A} V
| | | | | |- >> |{x,y:A}| = j2
| | | | | | | BY (SubstFor (h 5 p) ...) THEN (Lemma `doubleton_card`...)
| | | | | | |    THEN (E 17 THENM ReduceConcl...) THEN (ILeft...)
| | | | | | 13. x:|A|
| | | | | | 14. x ∈{A} V
| | | | | | 15. y:|A|
| | | | | | 16. y ∈{A} V
| | | | | | 17. ¬(λ z w. z=w in |A| ∨ E(z,w)(x,y))
| | | | | | 18. {x,y:A} ⊂ {A} V
| | | | | | 19. |{x,y:A}| = j2
| | | | | | 20. x@0:|A|
| | | | | | 21. y@0:|A|
| | | | | | 22. x@0 ∈{A} {x,y:A}
| | | | | | 23. y@0 ∈{A} {x,y:A}
| | | | | | 24. E(x@0,y@0)
| | | | | |- >> void
| | | | | | | BY (λf. f 23 THEN f 22(λi. HypChar `doubleton_char` i)...)
| | | | | | | 22. E(x@0,y@0)
| | | | | | | 23. y@0=x in |A| ∨ y@0=y in |A|
| | | | | | | 24. x@0=x in |A| ∨ x@0=y in |A|
| | | | | | |- >> void
| | | | | | | | BY (EThin 24 THEN EThin 23 ...)
| | | | | | | | 23. x@0=x in |A|
| | | | | | | | 24. y@0=x in |A|
| | | | | | | |- >> void
| | | | | | | | | BY (SubstForInHyp 'x@0=x in elts{A}(V)' 22!) THEN
| | | | | | | | |    (OnLastHyp (SubstForInHyp 'y@0=x in elts{A}(V)')!)
| | | | | | | | |    THEN (EOn 'x' 11!) THEN (Contradiction...)
| | | | | | | | 23. x@0=x in |A|
| | | | | | | | 24. y@0=y in |A|
| | | | | | | |- >> void
| | | | | | | | | BY (E 17 ...) THEN (IRight...) THEN
| | | | | | | | |    (SubstFor 'x=x@0 in elts{A}(V)'!) THEN
| | | | | | | | |    (SubstFor 'y=y@0 in elts{A}(V)'!)
| | | | | | | | 23. x@0=y in |A|
| | | | | | | | 24. y@0=x in |A|
| | | | | | | |- >> void
| | | | | | | | | BY (E 17 ...) THEN (IRight...) THEN
| | | | | | | | |    (SubstFor 'x=y@0 in elts{A}(V)'!) THEN
| | | | | | | | |    (SubstFor 'y=x@0 in elts{A}(V)'!) THEN
| | | | | | | | |    (BackThruHyp 10!)
| | | | | | | | 23. x@0=y in |A|
| | | | | | | | 24. y@0=y in |A|
| | | | | | | |- >> void
| | | | | | | | | BY (SubstForInHyp 'x@0=y in elts{A}(V)' 22!) THEN
| | | | | | | | |    (OnLastHyp (SubstForInHyp 'y@0=y in elts{A}(V)')!)
| | | | | | | | |    THEN (EOn 'y' 11!) THEN (Contradiction...)
| | 4. j2:int
| | 5. 2<j2
| | 6. ∃n:N+. n→ (j1,j2-1)
```

```
| |- >> ∃n:N+. n→ (j1,j2)
| | | BY (EOnThin 'j2' 3 THEN EThin 5 THEN EThin 5...)
| | | 3. j2:int
| | | 4. 2<j2
| | | 5. a4:N+
| | | 6. a4→ (j1,j2-1)
| | | 7. a5:N+
| | | 8. a5→ (j1-1,j2)
| | |- >> ∃n:N+. n→ (j1,j2)
| | | | BY (ITerm 'a4 + a5'...) THEN Try (E 7 THEN E 5 THEN MonoFromHyps `+`...*)
| | | | 9. G:Graph
| | | | 10. a4+a5≤ |V(G)|
| | | |- >> ∃s:FS(|G|). s ⊂ {|G|} V(G) & |s| = j1 & ∀x,y ϵ{|G|} s. ((¬(x=y in ||G||))->E(G)(x,y))
∨ |s| = j2 & ∀x,y ϵ{|G|} s. (¬(E(G)(x,y)))
| | | | | BY (BringHyp 10 THEN GraphE ``A V E`` 9...)
| | | | | 9. A:D
| | | | | 10. V:FS(A)
| | | | | 11. E:elts{A}(V)->elts{A}(V)->U1
| | | | | 12. x9:∀x,y:elts{A}(V). E(x,y) ∨ ¬(E(x,y))
| | | | | 13. x11:∀x,y:elts{A}(V). E(x,y) <=> E(y,x)
| | | | | 14. x12:∀x:elts{A}(V). ¬(E(x,x))
| | | | | 15. a4+a5≤ |V|
| | | | |- >> ∃s:FS(A). s ⊂ {A} V & |s| = j1 & ∀x,y ϵ{A} s. ((¬(x=y in |A|))->E(x,y)) ∨ |s|
= j2 & ∀x,y ϵ{A} s. (¬(E(x,y)))
| | | | | | BY InstLemma `pick` ['A';'V'] THEN Try (E 7 THEN E 5 THENS MonoFromHyps `+`...*)
| | | | | |    THEN OnLastHyp EThin
| | | | | | 16. v0:|A|
| | | | | | 17. v0 ϵ{A} V
| | | | | |- >> ∃s:FS(A). s ⊂ {A} V & |s| = j1 & ∀x,y ϵ{A} s. ((¬(x=y in |A|))->E(x,y)) ∨ |s|
= j2 & ∀x,y ϵ{A} s. (¬(E(x,y)))
| | | | | | | BY (InstLemma `fs_comp` ['A';'V -{A} v0';'E(v0)'] THENM Try Backchain...)
| | | | | | |    THEN Try (Ext...) THEN (Id!)
| | | | | | | 18. ∃r1,r2:FS(A). ∀x:elts{A}(V -{A} v0). x ϵ{A} r1 <=> x ϵ{A} V -{A} v0 & E(v0,x)
& x ϵ{A} r2 <=> x ϵ{A} V -{A} v0 & ¬(E(v0,x)) & disj{A}(r1,r2) & r1 ∪{A} r2 ={A} V -{A} v0
| | | | | | |- >> ∃s:FS(A). s ⊂ {A} V & |s| = j1 & ∀x,y ϵ{A} s. ((¬(x=y in |A|))->E(x,y)) ∨
|s| = j2 & ∀x,y ϵ{A} s. (¬(E(x,y)))
| | | | | | | | BY (RepeatProductE ``r1 r2`` 18 THENM Assert 'a4+a5-1≤ |r1|+|r2|'...)
| | | | | | | | 18. r1:FS(A)
| | | | | | | | 19. r2:FS(A)
| | | | | | | | 20. ∀x:elts{A}(V -{A} v0). x ϵ{A} r1 <=> x ϵ{A} V -{A} v0 & E(v0,x) & x ϵ{A} r2
<=> x ϵ{A} V -{A} v0 & ¬(E(v0,x))
| | | | | | | | 21. disj{A}(r1,r2)
| | | | | | | | 22. r1 ∪{A} r2 ={A} V -{A} v0
| | | | | | | | 23. |r1|+|r2|<(a4+a5)-1
| | | | | | | |- >> void
| | | | | | | | | BY (FLemma `union_card` [21] THENS FLemma `card_fnl` [22]
| | | | | | | | |    THENS InstLemma `fs_del_card` ['A';'V';'v0']...)
| | | | | | | | 18. r1:FS(A)
| | | | | | | | 19. r2:FS(A)
| | | | | | | | 20. ∀x:elts{A}(V -{A} v0). x ϵ{A} r1 <=> x ϵ{A} V -{A} v0 & E(v0,x) & x ϵ{A} r2
<=> x ϵ{A} V -{A} v0 & ¬(E(v0,x))
| | | | | | | | 21. disj{A}(r1,r2)
| | | | | | | | 22. r1 ∪{A} r2 ={A} V -{A} v0
| | | | | | | | 23. (a4+a5)-1≤ |r1|+|r2|
| | | | | | | |- >> ∃s:FS(A). s ⊂ {A} V & |s| = j1 & ∀x,y ϵ{A} s. ((¬(x=y in |A|))->E(x,y))
∨ |s| = j2 & ∀x,y ϵ{A} s. (¬(E(x,y)))
```

```
| | | | | | | | | BY (Assert 'a5≤ |r1| ∨ a4≤ |r2|' THENS OnLastHyp EThin...)
| | | | | | | | |    THENO ((Decide '|r1|<a5'...) THEN MonoFromHyps `-`) THEN Thin 23
| | | | | | | | | 23. a5≤ |r1|
| | | | | | | | | |- >> ∃s:FS(A). s ⊂ {A} V & |s| = j1 & ∀x,y ϵ{A} s. ((¬(x=y in |A|))->E(x,y))
∨ |s| = j2 & ∀x,y ϵ{A} s. (¬(E(x,y)))
| | | | | | | | | | BY (EOnThin '<A,<r1,<E,<x9,<x11,x12>>>>>' 8...)
| | | | | | | | | |    THEN OnLastHyp (UnfoldsInHyp ``Graph_carrier vertex edges``)
| | | | | | | | | |    THEN Try (Complete (RepeatExt!))
| | | | | | | | | | 8. A:D
| | | | | | | | | | 9. V:FS(A)
| | | | | | | | | | 10. E:elts{A}(V)->elts{A}(V)->U1
| | | | | | | | | | 11. x9:∀x,y:elts{A}(V). E(x,y) ∨ ¬(E(x,y))
| | | | | | | | | | 12. x11:∀x,y:elts{A}(V). E(x,y) <=> E(y,x)
| | | | | | | | | | 13. x12:∀x:elts{A}(V). ¬(E(x,x))
| | | | | | | | | | 14. a4+a5≤ |V|
| | | | | | | | | | 15. v0:|A|
| | | | | | | | | | 16. v0 ϵ{A} V
| | | | | | | | | | 17. r1:FS(A)
| | | | | | | | | | 18. r2:FS(A)
| | | | | | | | | | 19. ∀x:elts{A}(V -{A} v0). x ϵ{A} r1 <=> x ϵ{A} V -{A} v0 & E(v0,x) & x ϵ{A}
r2 <=> x ϵ{A} V -{A} v0 & ¬(E(v0,x))
| | | | | | | | | | 20. disj{A}(r1,r2)
| | | | | | | | | | 21. r1 ∪{A} r2 ={A} V -{A} v0
| | | | | | | | | | 22. a5≤ |r1|
| | | | | | | | | | 23. (a5≤ |r1|)->∃s:FS(A). s ⊂ {A} r1 & |s| = j1-1 & ∀x,y ϵ{A} s. ((¬(x=y
in |A|))->E(x,y)) ∨ |s| = j2 & ∀x,y ϵ{A} s. (¬(E(x,y)))
| | | | | | | | | | |- >> ∃s:FS(A). s ⊂ {A} V & |s| = j1 & ∀x,y ϵ{A} s. ((¬(x=y in |A|))->E(x,y))
∨ |s| = j2 & ∀x,y ϵ{A} s. (¬(E(x,y)))
| | | | | | | | | | | BY (EThin 23 ...) THEN Repeat (OnLastHyp EThin)
| | | | | | | | | | | 23. a16:FS(A)
| | | | | | | | | | | 24. a16 ⊂ {A} r1
| | | | | | | | | | | 25. |a16| = j1-1
| | | | | | | | | | | 26. ∀x,y ϵ{A} a16.((¬(x=y in |A|))->E(x,y))
| | | | | | | | | | | |- >> ∃s:FS(A). s ⊂ {A} V & |s| = j1 & ∀x,y ϵ{A} s. ((¬(x=y in |A|))->E(x,y))
∨ |s| = j2 & ∀x,y ϵ{A} s. (¬(E(x,y)))
| | | | | | | | | | | | BY ((ITerm 'a16 +{A} v0'...)!) THEN (ILeft...)
| | | | | | | | | | | | 27. a16 +{A} v0 ⊂ {A} V
| | | | | | | | | | | | |- >> |a16 +{A} v0| = j1
| | | | | | | | | | | | | BY (InstLemma `fs_add_card` ['A';'a16';'v0']...) THEN
| | | | | | | | | | | | |    (EOnThin 'v0' 24 THENS OnLastHyp EThin...) THEN (EThin 21...)
| | | | | | | | | | | | | 21. a5≤ |r1|
| | | | | | | | | | | | | 22. a16:FS(A)
| | | | | | | | | | | | | 23. |a16| = j1-1
| | | | | | | | | | | | | 24. ∀x,y ϵ{A} a16. ((¬(x=y in |A|))->E(x,y))
| | | | | | | | | | | | | 25. a16 +{A} v0 ⊂ {A} V
| | | | | | | | | | | | | 26. ∀A:D. ∀s:FS(A). ∀x:|A|. (¬(x ϵ{A} s))->(|s +{A} x| = |s|+1)
| | | | | | | | | | | | | 27. v0 ϵ{A} a16
| | | | | | | | | | | | | 28. v0 ϵ{A} r1
| | | | | | | | | | | | | 29. r1 ∪{A} r2 ⊂ {A} V -{A} v0
| | | | | | | | | | | | | 30. V -{A} v0 ⊂ {A} r1 ∪{A} r2
| | | | | | | | | | | | | |- >> void
| | | | | | | | | | | | | | BY (EOnThin 'v0' 29...) THEN (OnLastHyp EThin!)
| | | | | | | | | | | | | |    THEN (OnLastHyp (HypChar `fs_del_char`) THENS Contradiction...)
| | | | | | | | | | | | 27. a16 +{A} v0 ⊂ {A} V
| | | | | | | | | | | | 28. |a16 +{A} v0| = j1
```

```
| | | | | | | | | | | | 29. x:|A|
| | | | | | | | | | | | 30. y:|A|
| | | | | | | | | | | | 31. x ϵ{A} a16 +{A} v0
| | | | | | | | | | | | 32. y ϵ{A} a16 +{A} v0
| | | | | | | | | | | | 33. ¬(x=y in |A|)
| | | | | | | | | | | |- >> E(x,y)
| | | | | | | | | | | | | BY (Decide 'x=v0 in |A|' THEN Decide 'y=v0 in |A|'...)
| | | | | | | | | | | | | 34. x=v0 in |A|
| | | | | | | | | | | | | 35. y=v0 in |A|
| | | | | | | | | | | | |- >> E(x,y)
| | | | | | | | | | | | | | BY (E 33...)
| | | | | | | | | | | | | 34. x=v0 in |A|
| | | | | | | | | | | | | 35. ¬(y=v0 in |A|)
| | | | | | | | | | | | |- >> E(x,y)
| | | | | | | | | | | | | | BY (SubstFor 'x=v0 in elts{A}(V)'!) THEN (BackThruHyp 19!)
| | | | | | | | | | | | | 34. ¬(x=v0 in |A|)
| | | | | | | | | | | | | 35. y=v0 in |A|
| | | | | | | | | | | | |- >> E(x,y)
| | | | | | | | | | | | | | BY (BackThruHyp 12...) THEN
| | | | | | | | | | | | | |    (SubstFor 'y=v0 in elts{A}(V)'!) THEN (BackThruHyp 19!)
| | | | | | | | | | | | | 34. ¬(x=v0 in |A|)
| | | | | | | | | | | | | 35. ¬(y=v0 in |A|)
| | | | | | | | | | | | |- >> E(x,y)
| | | | | | | | | | | | | | BY (BackThruHyp 26!)
| | | | | | | | | | | 23. a16:FS(A)
| | | | | | | | | | | 24. a16 ⊂ {A} r1
| | | | | | | | | | | 25. |a16| = j2
| | | | | | | | | | | 26. ∀x,y ϵ{A} a16. (¬(E(x,y)))
| | | | | | | | | | |- >> ∃s:FS(A). s ⊂ {A} V & |s| = j1 & ∀x,y ϵ{A} s. ((¬(x=y in |A|))->E(x,y))
∨ |s| = j2 & ∀x,y ϵ{A} s. (¬(E(x,y)))
| | | | | | | | | | | | BY ((ITerm 'a16'...)!) THEN (IRight...)
| | | | | | | | | 23. a4≤ |r2|
| | | | | | | | |- >> ∃s:FS(A). s ⊂ {A} V & |s| = j1 & ∀x,y ϵ{A} s. ((¬(x=y in |A|))->E(x,y))
∨ |s| = j2 & ∀x,y ϵ{A} s. (¬(E(x,y)))
| | | | | | | | | | BY (EOnThin '<A,<r2,<E,<x9,<x11,x12>>>>>' 6...)
| | | | | | | | | |    THEN OnLastHyp (UnfoldsInHyp ``Graph_carrier vertex edges``)
| | | | | | | | | |    THEN Try (Complete (RepeatExt!))
| | | | | | | | | | 6. a5:N+
| | | | | | | | | | 7. a5→ (j1-1,j2)
| | | | | | | | | | 8. A:D
| | | | | | | | | | 9. V:FS(A)
| | | | | | | | | | 10. E:elts{A}(V)->elts{A}(V)->U1
| | | | | | | | | | 11. x9:∀x,y:elts{A}(V). E(x,y) ∨ ¬(E(x,y))
| | | | | | | | | | 12. x11:∀x,y:elts{A}(V). E(x,y) <=> E(y,x)
| | | | | | | | | | 13. x12:∀x:elts{A}(V). ¬(E(x,x))
| | | | | | | | | | 14. a4+a5≤ |V|
| | | | | | | | | | 15. v0:|A|
| | | | | | | | | | 16. v0 ϵ{A} V
| | | | | | | | | | 17. r1:FS(A)
| | | | | | | | | | 18. r2:FS(A)
| | | | | | | | | | 19. ∀x:elts{A}(V -{A} v0). x ϵ{A} r1 <=> x ϵ{A} V -{A} v0 & E(v0,x) & x ϵ{A}
r2 <=> x ϵ{A} V -{A} v0 & ¬(E(v0,x))
| | | | | | | | | | 20. disj{A}(r1,r2)
| | | | | | | | | | 21. r1 ∪{A} r2 ={A} V -{A} v0
| | | | | | | | | | 22. a4≤ |r2|
```

```
| | | | | | | | | | 23. (a4≤ |r2|)->∃s:FS(A). s ⊂ {A} r2 & |s| = j1 & ∀x,y ϵ{A} s. ((¬(x=y
in |A|))->E(x,y)) ∨ |s| = j2-1 & ∀x,y ϵ{A} s. (¬(E(x,y)))
| | | | | | | | | |- >> ∃s:FS(A). s ⊂ {A} V & |s| = j1 & ∀x,y ϵ{A} s. ((¬(x=y in |A|))->E(x,y))
∨ |s| = j2 & ∀x,y ϵ{A} s. (¬(E(x,y)))
| | | | | | | | | | | BY (EThin 23...) THEN Repeat (OnLastHyp EThin)
| | | | | | | | | | | 23. a16:FS(A)
| | | | | | | | | | | 24. a16 ⊂ {A} r2
| | | | | | | | | | | 25. |a16| = j1
| | | | | | | | | | | 26. ∀x,y ϵ{A} a16. ((¬(x=y in |A|))->E(x,y))
| | | | | | | | | | |- >> ∃s:FS(A). s ⊂ {A} V & |s| = j1 & ∀x,y ϵ{A} s. ((¬(x=y in |A|))->E(x,y))
∨ |s| = j2 & ∀x,y ϵ{A} s. (¬(E(x,y)))
| | | | | | | | | | | | BY ((ITerm 'a16'...)!) THEN (ILeft...)
| | | | | | | | | | | 23. a16:FS(A)
| | | | | | | | | | | 24. a16 ⊂ {A} r2
| | | | | | | | | | | 25. |a16| = j2-1
| | | | | | | | | | | 26. ∀x,y ϵ{A} a16. (¬(E(x,y)))
| | | | | | | | | | |- >> ∃s:FS(A). s ⊂ {A} V & |s| = j1 & ∀x,y ϵ{A} s. ((¬(x=y in |A|))->E(x,y))
∨ |s| = j2 & ∀x,y ϵ{A} s. (¬(E(x,y)))
| | | | | | | | | | | | BY ((ITerm 'a16 +{A} v0'...)!) THEN (IRight...)
| | | | | | | | | | | | 27. a16 +{A} v0 ⊂ {A} V
| | | | | | | | | | | |- >> |a16 +{A} v0| = j2
| | | | | | | | | | | | | BY (InstLemma `fs_add_card` ['A';'a16';'v0']...) THEN
| | | | | | | | | | | | |    (EOnThin 'v0' 24 THENS OnLastHyp EThin...) THEN
| | | | | | | | | | | | |    (EThin 21...)
| | | | | | | | | | | | | 21. a4≤ |r2|
| | | | | | | | | | | | | 22. a16:FS(A)
| | | | | | | | | | | | | 23. |a16| = j2-1
| | | | | | | | | | | | | 24. ∀x,y ϵ{A} a16. (¬(E(x,y)))
| | | | | | | | | | | | | 25. a16 +{A} v0 ⊂ {A} V
| | | | | | | | | | | | | 26. ∀A:D. ∀s:FS(A). ∀x:|A|. (¬(x ϵ{A} s))->(|s +{A} x| = |s|+1)
| | | | | | | | | | | | | 27. v0 ϵ{A} a16
| | | | | | | | | | | | | 28. v0 ϵ{A} r2
| | | | | | | | | | | | | 29. r1 ∪{A} r2 ⊂ {A} V -{A} v0
| | | | | | | | | | | | | 30. V -{A} v0 ⊂ {A} r1 ∪{A} r2
| | | | | | | | | | | | |- >> void
| | | | | | | | | | | | | | BY (EOnThin 'v0' 29 ...) THEN (OnLastHyp EThin!) THEN
| | | | | | | | | | | | | |    (OnLastHyp (HypChar `fs_del_char`) THENS Contradiction...)
| | | | | | | | | | | | 27. a16 +{A} v0 ⊂ {A} V
| | | | | | | | | | | | 28. |a16 +{A} v0| = j2
| | | | | | | | | | | | 29. x:|A|
| | | | | | | | | | | | 30. y:|A|
| | | | | | | | | | | | 31. x ϵ{A} a16 +{A} v0
| | | | | | | | | | | | 32. y ϵ{A} a16 +{A} v0
| | | | | | | | | | | | 33. E(x,y)
| | | | | | | | | | | |- >> void
| | | | | | | | | | | | | BY (Decide 'x=v0 in |A|' THEN Decide 'y=v0 in |A|'...)
| | | | | | | | | | | | |    THEN BringHyp 33
| | | | | | | | | | | | | 34. x=v0 in |A|
| | | | | | | | | | | | | 35. y=v0 in |A|
| | | | | | | | | | | | |- >> ¬(E(x,y))
| | | | | | | | | | | | | | BY SubstFor 'x=v0 in elts{A}(V)' THENM SubstFor 'y=v0 in elts{A}(V)'
| | | | | | | | | | | | | |    THEN (Try Backchain!)
| | | | | | | | | | | | | 34. x=v0 in |A|
| | | | | | | | | | | | | 35. ¬(y=v0 in |A|)
| | | | | | | | | | | | |- >> ¬(E(x,y))
```

```
| | | | | | | | | | | | | | | | BY SubstFor 'x=v0 in elts{A}(V)' THEN (Try (BackThruHyp 19)!)
| | | | | | | | | | | | | | | 34. ¬(x=v0 in |A|)
| | | | | | | | | | | | | | | 35. y=v0 in |A|
| | | | | | | | | | | | | | |- >> ¬(E(x,y))
| | | | | | | | | | | | | | | | BY (Assert 'E(v0,x)' THENO BackThruHyp 12 THENO SubstFor 'v0=y in
elts{A}(V)'!)
| | | | | | | | | | | | | | | | 36. E(v0,x)
| | | | | | | | | | | | | | | | 37. E(x,y)
| | | | | | | | | | | | | | | |- >> void
| | | | | | | | | | | | | | | | | BY BringHyp 36 THEN (BackThruHyp 19!)
| | | | | | | | | | | | | | | 34. ¬(x=v0 in |A|)
| | | | | | | | | | | | | | | 35. ¬(y=v0 in |A|)
| | | | | | | | | | | | | | |- >> ¬(E(x,y))
| | | | | | | | | | | | | | | | BY (BackThruHyp 26!)
```