Formalizing the Excluded Minor Characterization of Binary Matroids in the Lean Theorem Prover

by

Alena Gusakov

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Combinatorics & Optimization

Waterloo, Ontario, Canada, 2024

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

A matroid is a mathematical object that generalizes the notion of linear independence of a set of vectors to an abstract independence of sets, with applications to optimization, linear algebra, graph theory, and algebraic geometry. Matroid theorists are often concerned with representations of matroids over fields. Tutte's seminal theorem proven in 1958 characterizes matroids representable over $GF(2)$ by noncontainment of $U_{2,4}$ as a matroid minor. In this thesis, we document a formalization of the theorem and its proof in the Lean Theorem Prover, building on its community-built mathematics library, `mathlib`.

Acknowledgements

I'm deeply grateful to my advisor Peter Nelson, both for giving me the opportunity to work on this exciting and unconventional project and for supporting me through all of the difficulties along the way. Thank you for taking a chance on me. I look forward to collaborating with you more in the future.

Many thanks to Felix Lazebnik, Sebastian Cioaba, and Robert Coulter for their advice and words of support when I was an undergraduate interested in pursuing research. Each professor provided me with perspectives on research and graduate school that helped me become more confident and resilient in the face of setbacks. I especially appreciate Dr. Lazebnik keeping in touch and being someone I could reach out to during the more difficult parts of my degree. I hope you have a happy retirement and I wish you and your family well.

I'm very grateful to each of my readers, David Jao and Jonathan Leake, for agreeing to read my thesis and providing timely and helpful comments.

I would like to thank my parents for doing everything in their power to help me despite the distance. From driving for ten hours one way just to help me move apartments to taking care of my cat Serik even though I know he's a handful, you guys go above and beyond to make things easier to me and I feel so loved and supported.

I would like to thank my younger brother Raymond for being someone I can count on to keep in touch when we can. We may not get to talk often these days, but I cherish the occasional several-hour-long phone call when we catch up and laugh at all the ridiculous circumstances we find ourselves in. Good luck on wrapping up undergrad.

I would like to thank my friends from home for staying in touch and hanging out whenever we are all back in town. Thank you to Jimmy, Joy, Peyton, Jared, Hewitt, Claire, Alex, and Logan. I'm really grateful to have all of you in my life. In particular, I'm grateful for our multiple D&D campaigns, some of which have been going on for over two years (RIP Qikki Akala), which have given me something to look forward to every week throughout my degree.

I would also like to thank the friends that I've made since coming to Waterloo. Thanks to Ed, Brian (even though you can't skate), Manya, and Rish. In particular, I'd like to thank Rish for inspiring my obsession for salsa dancing. What started out as something I just casually tagged along for turned into an exciting hobby that has given me a sense of community and belonging. I'll never stop dancing again.

Dedication

I am dedicating this thesis to my maternal grandmother who lives in Ukraine, who helped raise me and my brother, and whom I will see for the first time in over ten years in a few days.

Table of Contents

Chapter 1

Introduction

In this thesis, we document the formalization of the excluded minor characterization of binary matroids in Lean, as well as the development of a proof assistant library for representable matroids. We also provide groundwork for the future formalization of other excluded minor characterizations, most notably the excluded minor characterizations of representable matroids and graphic matroids. The excluded minor for binary matroids, $U_{2,4}$, is naturally an excluded minor for regular matroids (since they must be representable over all fields), and graphic matroids (as they must be regular matroids) [25].

1.1 Proof Assistants

A proof assistant is a software program that is used to facilitate the writing of formal proofs that can then be verified by computers. We refer to the process of translating mathematical definitions and proofs into a proof assistant's language as *formalization*.

Why do we want to formalize mathematics? There are numerous perspectives and answers to this question[1][2]. However, the author of this thesis has a more personal reason.

Despite having a love for mathematics, the author finds it difficult to work on traditional pen-and-paper proofs due to ADHD. However, they do not face the same difficulty when it comes to programming. The author believes that the formality and strictness of programs make them easier to reason with compared to the implicit assumptions and conclusions

[1] https://xenaproject.wordpress.com/what-is-the-xena-project/
[2] https://proofassistants.stackexchange.com/questions/57/usage-and-importance-of-proof-assistants

often found in mathematics. The process of proving every detail in a proof assistant, while tedious, forces one to deeply understand the material. This often provides us with new insights and allows us to catch mistakes that might otherwise be overlooked. This granular breakdown of proofs also makes them more accessible, as someone learning the material can look into details of a proof that might typically be glossed over in a course or a paper.

Additionally, the author believes that the feedback received from a program compiling counteracts their difficulties with focus, as one can easily spend hours on a programming project without noticing the time passing. The author is optimistic that their work on this thesis, and future formalization projects, will contribute to a future of proof assistants that are sophisticated enough to write proofs as seamlessly as by hand, thus making mathematics more accessible while simultaneously ensuring correctness and revealing new insights.

The use of computers in proving mathematical theorems has a history dating back to as early as the late 1960's, from Nicolaas Govert de Bruijn et al.'s *Automath* [9], in which novel concepts such as typed lambda calculus were first seen, to Michael Gordon et al.'s *Logic for Computable Functions (LCF)* [16], a progenitor to HOL and Isabelle. Since then, a number of theorem provers with different underlying systems have cropped up.

1.2 The Lean Theorem Prover

The formalization project outlined in this thesis was completed using the Lean 3 Theorem Prover[3], developed by Leonardo de Moura at Microsoft Research. Lean has several features that make it a desirable tool for mathematics formalization, as well as a large user base of mathematicians. An important aspect of Lean is its community-built mathematics library, `mathlib` [23].

Our formalization project uses libraries for linear algebra and abstract algebra from `mathlib`. Shortly before this project began, Lean 4 was released and massive effort was put into porting `mathlib` from Lean 3 to Lean 4. The formalization project outlined in this paper will likely be one of the final projects to be ported to Lean 4, with the goal of contributing its results to `mathlib4`.

In recent years, Lean has been used in notable formalization projects such as the liquid tensor experiment [7] and the sphere eversion project [32]. In November 2023, Terence Tao announced a project [28] on formalizing the proof by Gowers et al. of the Polynomial

[3] https://leanprover-community.github.io/lean3/

Freiman-Ruzsa (PFR) conjecture over $\mathbb{F}_2$ [17]. This project was completed in less than a month [29], as the high-profile nature of it attracted a large number of contributors from the `mathlib` community.

As proof assistants grow in popularity among mathematics communities, the number of fields and subfields of mathematics that they are applied to grows by the day. With applications from optimization to linear algebra to graph theory, it's reasonable for matroid theory to be added to the list.

1.3 Combinatorics & Matroid Theory

One of the first notable milestones in combinatorics formalization was the use of Coq by Georges Gonthier and Benjamin Werner [14] in formalizing the proof of the Four-Color Theorem by Robertson et al. [27]. Since then, numerous combinatorics libraries have evolved in various proof assistants. Christian Doczkal and Damien Pous have done extensive work to build a graph theory library in Coq that covers topics such as minors and treewidth [12]. Szemeredi's Regularity Lemma has recently made an appearance in multiple proof assistants - from the formalization in Lean by Bhavik Mehta and Yael Dillies [11], to the simultaneous formalization in Isabelle/HOL by Edmonds et al [13]. The Kruskal-Katona theorem also has a formalization in Lean by Bhavik Mehta [24].

Matroid theory pre-dates computer-assisted proofs by about thirty years [33][1], which is not much in the context of mathematical history. Matroids were originally created as a way of generalizing the notion of linear independence of a set of vectors to an abstract independence of sets. Matroids have applications in linear algebra, graph theory, optimization [8], and algebraic geometry [19]. For a comprehensive reference on their combinatorial theory, see Oxley [25].

Some early matroid formalization examples include [6], which follows some basic definitions from [25], and [21], which describes a tactic for Coq that uses matroid axioms for projective geometry.

1.4 Excluded Minor for Binary Matroids

Matroid theorists are interested in the concept of representability in matroids, which we will define later in this thesis. To determine if a matroid can be represented over a specific field $\mathbb{F}$, a common approach is to identify the minimal obstacles to $\mathbb{F}$-representability.

Because a matroid can only be $\mathbb{F}$-representable if its minors are $\mathbb{F}$-representable, we can characterize $\mathbb{F}$-representability by listing minors that prevent it. These are called *excluded minors* for $\mathbb{F}$-representability, and finding excluded minors for an arbitrary field $\mathbb{F}$ is an open problem.

However, $GF(2)$ is a field for which representability is well understood. This thesis documents our formalization of Tutte's seminal theorem in the 1950s that characterizes matroids representable over $GF(2)$ by their excluded minors. [30]. We provide the statement of the theorem.

Theorem 1.4.1 (Tutte's 1958 excluded minor characterization of binary matroids)**.** *A matroid is binary if and only if it has no $U_{2,4}$-minor.*

1.5 This Thesis

The organization of this thesis is as follows. Section 1 covers history and outlines the proof of the excluded minor characterization of binary matroids, Section 2 details mathematical definitions and design choices for our formalization, and Section 3 presents our formalized proof of the theorem in `lean`.

The goal of this thesis is to provide an exposition on the Tutte excluded minor theorem and our formalization of it, which is based on a matroid API written by Peter Nelson[4]. While we do not explain every line of the code, we attempt to guide the reader through the broad strokes and how the code corresponds to the mathematics. Sections 2.1 and 2.2 are a discussion of Nelson's matroid formalization, and everything from Section 2.3 onward is original work by the author (all code concerning matroid representations). A repository with the minimum set of definitions and lemmas required for the theorem can be found on Github[5].

[4] https://github.com/apnelson1/lean-matroids
[5] https://github.com/agusakov/excluded_minor_binary

Chapter 2

Implementation

In the process of developing this implementation of Tutte's theorem, we had to make careful use of lemmas from the `mathlib` libraries[1] for linear algebra, fields (specifically $GF(2)$, or `zmod 2` as it's referred to in lean), constructions such as `finsupp` (more on this later), and a lot of set theory API[2]. We also rely on a matroid API written by Peter Nelson.

2.1 Matroid Definitions

Before one can begin to formalize a topic, one must decide on a definition. Mathematicians typically are comfortable switching between equivalent definitions as needed, but when it comes to formalization, this is easier said than done. Type theoretically, there are usually multiple ways of defining an object, each with its own strengths and weaknesses when it comes to their usability. In addition, matroids have several "cryptomorphic" mathematical definitions [25], which compounds the problem of choosing a definition.

2.1.1 Cryptomorphic Definitions

Matroids can be defined with respect to their various substructures, such as (but not limited to) independent sets [25, p.7], circuits [25, p.10], closure [25, p.27], rank function [25, p.23],

[1] https://github.com/leanprover-community/mathlib

[2] In the world of software, the term 'API' often refers to a library of functions used to interact with a piece of software at the programmatic level. In the context of theorem provers, it is standard to use the term analogously, referring to a library of definitions and basic lemmas used to interact with mathematical objects.

and flats [25, p.31]. Considerations for possible formal definitions of an object include generality, compatibility with other objects, and equivalences with other definitions. For this reason, we decided on the primary definition being one that invokes bases. The base definition has the advantage of having fewer axioms to prove, meaning that showing that something is a matroid requires fewer steps. In addition, it is easy to demonstrate its equivalence to another common definition that uses independence axioms.

A matroid $M = (E, \mathcal{B})$ is a set E and a nonempty collection $\mathcal{B}$ of subsets of E that satisfies an exchange property axiom, which we will outline shortly. This formalization project builds on an existing matroid API written by Peter Nelson, which is designed to be as general as possible, and hence allows for infinite matroids. Allowing matroids to be infinite presents some surprising difficulties, even at the level of merely defining them. In fact, until 2013, the question of which axioms are needed in order to properly define infinite matroids was something of an open problem, with contemporary definitions interfering with common matroid operations such as duality. In Bruhn et al.'s paper [4], five equivalent definitions are proposed for infinite matroids, which give us the correct notion of duality, as well as allow for applications such as matroid duals of vector spaces and infinite graphs. For our implementation, we use Bruhn et al's infinite axiom for the base definition.

Definition 1. *A matroid $M = (E, \mathcal{B})$ is a set E and a nonempty collection $\mathcal{B}$ of subsets of E such that*

(B0) *$\mathcal{B}$ is non-empty*

(B1) ***Exchange Property:*** *For all $B_1, B_2 \in \mathcal{B}$, and for all $a \in B_1 \backslash B_2$, there exists $b \in B_2 \backslash B_1$ such that $(B_1 \backslash \{a\}) \cup \{b\} \in \mathcal{B}$*

(B2) ***Maximality:*** *For all $X \subseteq E(M)$ and $I \subseteq X$, such that there exists $B_1 \in B$ with $I \subseteq B_1$, there exists a subset $Y \subseteq X$ which is maximal with respect to the property "there exists $B_2 \in \mathcal{B}$ such that $Y \subseteq B_2$".*

Here, we use the notational convention $E(M)$ to denote the ground set of matroid M. When M has ground set E, we use E and $E(M)$ interchangeably. Furthermore, an element of $\mathcal{B}$ is called a *base*.

Now, readers may recognize that matroids are more commonly defined with respect to their independent sets. This is the definition that Oxley introduces matroids with [25], and more importantly, it is a definition that we use for our third definition of a representation, so we provide a definition of independence here.

Definition 2. *Let $I \subseteq E(M)$. We say I is independent in M if there exists some base $B \in \mathcal{B}$ such that $I \subseteq B$. We also say a set I is dependent if it is not independent.*

We can derive an equivalent definition of matroids using independent sets (also allowing infinite matroids [4]).

Definition 3. *A matroid $M = (E, \mathcal{I})$ is a set E and a collection $\mathcal{I}$ of subsets of E such that*

(I0) The empty set $\varnothing$ is in $\mathcal{I}$.

(I1) For all $I, J \subseteq E(M)$, if $J \in \mathcal{I}$ and $I \subseteq J$, then $I \in \mathcal{I}$.

(I2) For all $I, B \subseteq E(M)$, if $I \in \mathcal{I}$ and I is not a maximally independent set of M, and $B \in \mathcal{I}$ is maximally independent in M, then there exists some $x \in B \backslash I$ such that $I \cup \{x\} \in \mathcal{I}$.

*(I3) **Maximality:** There exists some $I \in \mathcal{I}$ such that I is maximal with respect to set inclusion.*

Then the collection of subsets $\mathcal{I}$ of E is precisely the set of independent sets of M. This definition allows us to avoid working with cardinalities of sets, but sometimes that is what we need, so for that reason we define finite-rank matroids as follows:

Definition 4. *A finite-rank matroid $M = (E, \mathcal{I})$ is a set E and a collection $\mathcal{I}$ of subsets of E such that*

(I0) The empty set $\varnothing$ is in $\mathcal{I}$.

(I1) For all $I, J \subseteq E(M)$, if $J \in \mathcal{I}$ and $I \subseteq J$, then $I \in \mathcal{I}$.

(I2) For all $I, B \subseteq E(M)$, if $I \in \mathcal{I}$ and I is not a maximally independent set of M, and $B \in \mathcal{I}$ is maximally independent in M, then there exists some $x \in B \backslash I$ such that $I \cup \{x\} \in \mathcal{I}$.

*(I3') **Bounded:** There exists some $n \in \mathbb{N}$ such that, for all $I \in \mathcal{I}$, $|I| \leq n$.*

Again, readers may notice that this still is not the typical definition of matroids, but it is an intermediate step between the first definition and the one we actually need. The following is equivalent to definition 4:

Definition 5. *A finite-rank matroid $M = (E, \mathcal{I})$ is a set E and a collection $\mathcal{I}$ of subsets of E such that*

(I0) The empty set $\varnothing$ is in $\mathcal{I}$.

(I1) For all $I, J \subseteq E(M)$, if $J \in \mathcal{I}$ and $I \subseteq J$, then $I \in \mathcal{I}$.

(I2') For all $I, J \in \mathcal{I}$, if $|I| < |J|$, then there exists some $e \in J \backslash I$ such that $I \cup \{e\} \in \mathcal{I}$.

*(I3') **Bounded:** There exists some $n \in \mathbb{N}$ such that, for all $I \in \mathcal{I}$, $|I| \leq n$.*

One last important thing to note is that matroid theorists overload the term "basis" to mean two things. Our definition of "base" is what matroid theorists often refer to as a basis, but the other possible meaning is a basis for a set, i.e. if $X \subseteq E(M)$, we say B is a basis of X if B is a maximal independent subset of X. For the sake of unambiguity within our code, we decided to differentiate between a basis of a matroid and a basis of a subset of the ground set by creating two separate definitions. A basis of a matroid, $B \in \mathcal{B}$, will henceforth be referred to as a `base` in this thesis, while a basis of a subset of the ground set will be a `basis`.

2.1.2 Lean Definitions

We first define predicates that correspond to the exchange property and maximality axioms we have seen above. This is the first time we show formal `lean` code, so note that `(P : set α → Prop)` is an arbitrary predicate that may or may not be true for a term of type `set α`. We annotate the code using built-in delimiters */-- docstring -/* (for official documentation strings) and */- comment -/*, or for inline comments, *-- comment*. The comments are not part of the code itself.

```
/-- A predicate `P` on sets satisfies the exchange property if, for all `
   X` and `Y` satisfying `P` and all `a ∈ X \ Y`, there exists `b ∈ Y \ X`
   so that swapping `a` for `b` in `X` maintains `P`.-/
def exchange_property (P : set α → Prop) : Prop :=
  ∀ X Y, P X → P Y → ∀ a ∈ X \ Y, ∃ b ∈ Y \ X, P (insert b (X \ {a}))
-- This corresponds to axiom (B0)

/-- A predicate `P` on sets satisfies the maximal subset property if, for
   all `X` containing some `I` satisfying `P`, there is a maximal subset of
   `X` satisfying `P`. -/
def exists_maximal_subset_property (P : set α → Prop) (X : set α) :
```

```
    Prop :=
  ∀ I, P I → I ⊆ X → (maximals (⊆) {Y | P Y ∧ I ⊆ Y ∧ Y ⊆ X}).nonempty
-- This corresponds to axiom (B1)
```

For our definition of a matroid, `P` will effectively be our `base` predicate, i.e. `base : set α → Prop` is true or false for any `set α`. Then we can define matroids as follows:

```
/-- A `matroid` is a nonempty collection of sets satisfying the exchange
    property and the maximal subset property. Each such set is called a `
    base` of the matroid. -/
structure matroid (α : Type*) :=
  (ground : set α)
  (base : set α → Prop)
  (exists_base' : ∃ B, base B)
  (base_exchange' : exchange_property base)
  (maximality : ∀ X ⊆ ground, exists_maximal_subset_property (λ I, ∃ B,
   base B ∧ I ⊆ B) X)
  (subset_ground' : ∀ B, base B → B ⊆ ground)
```

From our main definition, the implementation of independence and bases of subsets of the ground set was fairly straightforward:

```
/-- A set is independent if it is contained in a base.  -/
def indep (M : matroid α) (I : set α) : Prop := ∃ B, M.base B ∧ I ⊆ B

/-- A subset of `M.E` is dependent if it is not independent . -/
def dep (M : matroid α) (D : set α) : Prop := ¬M.indep D ∧ D ⊆ M.E

/-- A basis for a set `X ⊆ M.E` is a maximal independent subset of `X`
  (Often in the literature, the word 'basis' is used to refer to what we
   call a 'base'). -/
def basis (M : matroid α) (I X : set α) : Prop :=
  I ∈ maximals (⊆) {A | M.indep A ∧ A ⊆ X} ∧ X ⊆ M.E
```

As one can observe, the definition of an independent set is simply a set that is contained in some `base`, and our definition of a `basis` mirrors the `exists_maximal_subset_property` we defined earlier. In essence, if $X \subseteq E(M)$, a `basis` is a maximal subset $I \subseteq X$ such that $I \subseteq B$ for some `base` $B \in \mathcal{B}$.

Now, while these two definitions gave us the ability to talk about independent sets and bases of a set, we needed to also be able to define a matroid using independence. The following is one possible implementation:

```
/-- A version of the independence axioms that avoids cardinality -/
def matroid_of_indep (E : set α) (indep : set α → Prop)
(h_empty : indep ∅)
(h_subset : ∀ {I J}, indep J → I ⊆ J → indep I)
(h_aug : ∀{I B}, indep I → I ∉ maximals (⊆) indep → B ∈ maximals (⊆)
   indep →
  ∃ x ∈ B \ I, indep (insert x I))
(h_maximal : ∀ X ⊆ E, exists_maximal_subset_property indep X)
(h_support : ∀ I, indep I → I ⊆ E) :
  matroid α :=
matroid_of_base E (λ B, B ∈ maximals (⊆) indep)
```

and our lean implementation looks like:

```
/-- If there is an absolute upper bound on the size of an independent
   set, then the maximality axiom isn't needed to define a matroid by
   independent sets. -/
def matroid_of_indep_of_bdd (E : set α) (indep : set α → Prop)
/- Empty set is independent -/
(h_empty : indep ∅)
/- Every subset of an independent set is independent -/
(h_subset : ∀ {I J}, indep J → I ⊆ J → indep I)
/- For all I, B ⊆ E(M), if I is independent and I is not a maximally
   independent set of M, and B is maximally independent in M, then
   there exists some x ∈ B \ I such that I ∪ {x} is independent. -/
(h_aug : ∀{I B}, indep I → I ∉ maximals (⊆) indep → B ∈ maximals (⊆)
   indep →
  ∃ x ∈ B \ I, indep (insert x I))
/- There exists some n ∈ N such that, for all indep I, |I| ≤ n. This
   corresponds to bounded axiom `(I3)'`. -/
(h_bdd : ∃ n, ∀ I, indep I → I.finite ∧ I.ncard ≤ n)
(h_support : ∀ I, indep I → I ⊆ E) : matroid α :=
matroid_of_indep E indep h_empty h_subset h_aug
  (λ X h, exists_maximal_subset_property_of_bounded h_bdd X)
  h_support
```

The following is our implementation of this definition, building on and invoking all of our previous definitions:

```
/-- Alternate definition that uses the independence axioms that are
   typically used -/
def matroid_of_indep_of_bdd' (E : set α) (indep : set α → Prop)
(h_empty : indep ∅)
```

```
(h_subset : ∀ {⦃I J⦄}, indep J → I ⊆ J → indep I)
(ind_aug : ∀ {⦃I J⦄}, indep I → indep J → I.ncard < J.ncard →
  ∃ e ∈ J, e ∉ I ∧ indep (insert e I))
(h_bdd : ∃ n, ∀ I, indep I → I.finite ∧ I.ncard ≤ n)
(h_support : ∀ I, indep I → I ⊆ E) : matroid α :=
matroid_of_indep_of_bdd E indep h_empty h_subset [...] h_bdd h_support
```

where the `[...]` is a placeholder for the proof that `ind_aug` lends itself to the correct augmentation axiom that we have in `matroid_of_indep_of_bdd`. This allows us to define a matroid by defining its independent sets. For instance, this is how we could define a matroid on ground set $\{1, 2, 3, 4, 5\}$ where the only independent set is the empty set:

```
def rk_two_matroid : matroid ℕ :=
    matroid_of_indep_of_bdd'
    /- The ground set is {1,2,3,4,5} -/
    {1,2,3,4,5}
    /- A set is independent iff its cardinality is at most 2 -/
    (λ I ⊆ {1,2,3,4,5}, I.ncard ≤ 2)
    /- Proofs that the axioms hold -/
    [...]
```

This last matroid definition, `matroid_of_indep_of_bdd'`, is what we will use to give ourselves some flexibility in defining matroid representations later.

2.1.3 Dependent Type Theory

`Lean` is written using dependent type theory, an alternative to ZFC. This meant that there were type theoretic decisions we had to make before we could implement the definition. We will briefly discuss basic concepts, but for a more comprehensive overview, see [2] (or for the Lean 4 version, [3]. The content is roughly the same, just with different Lean syntax).

The general idea behind dependent type theory is that every object is either a type, or a term associated with some type. In `lean`, we can use the `#check` command to see what type an object is. As described in the previous section, we use comment delimiters `/-- -/`, `/- -/`, and `--` for annotations, and these are not the actual `lean` code.

```
/- This will output that `2` is a term of type `ℕ` -/
#check 2
--output: 2 : ℕ

open_locale real
```

```
/- This will output that `π` is a term with type `ℝ` -/
#check π
-- output: π : ℝ

/- This will output that `{3, 4, π}` is a term with type `set ℝ` -/
#check ({3, 4, π} : set ℝ)
```

If we want to define a type, we can use `universe` to give it a type universe, or define an arbitrary type in `*`. The distinction is outside the scope of this thesis, but here is how we can define an arbitrary type α:

```
/- This says `α` is a `Type` in an arbitrary universe -/
variables (α : Type*)

/- This will output `Type*` -/
#check α
-- output: `α : Type*`
```

Mathematically, a matroid is a structure on a ground set E, similar to how a group or a ring is a structure on an underlying set. It might seem natural to mirror the approach `mathlib` takes for groups and rings, where the ground set in question is viewed as a type. Thus, a matroid can be defined as a structure on a "ground" type `E`. This has certain advantages. For example, subsets `A` and `B` of the ground set would correspond to terms `A : set E` and `B : set E`, respectively. The intersection `A ∩ B` would also have type `set E`, as intersection in `lean` is defined as a function that takes two arguments `A B : set E` and outputs a term of type `set E`. More generally, the fact that subsets of `E` are closed under natural mathematical operations such as union, intersection, difference, etc. is seamlessly captured using the type system.

Despite these advantages, we take a different approach. Instead of considering the ground set as a type, we introduce an underlying type α (which can be thought of as the type of all "possible matroid elements"), and view the ground set itself as a term `E` of type `set α`. This approach does not have the nice properties discussed earlier. Subsets `A` and `B` of `E` would correspond to terms `A : set α` and `B : set α`, together with proofs that `A ⊆ E` and `B ⊆ E`. Although `A ∩ B ⊆ E` is still true, now it requires a proof. This proof is trivial, but is no longer as seamless as before.

However, our approach has its own substantial advantages, because it allows us to circumvent the inflexible nature of the type system when dealing with different matroids on related ground sets. For example, consider the matroids M and $M\backslash e$. Mathematically,

it is trivial to see that $E(M\backslash e) = E(M)\backslash\{e\}$. However, if the ground sets are defined as types, i.e. `M` has ground type `α` for some `α`, then when we delete an element `e : α` of `M`, we are forced to construct a new type `β` for `M \ e`. This is problematic, as `α = β` is not a valid statement. One would need to establish an equivalence between the types, `α ≃ β`, which is unwieldy.

In Lean, there is one type that is quite important for a proof-writing environment. We have type `Prop` that represents propositions [2], and allows us to make assertions about mathematical objects. `Prop` can be thought of as having values "true" or "false".

```
/-- The proposition that 6 is less than or equal to a natural number
    `n : ℕ`. This can be true or false depending on the value of `n`. -/
def six_le (n : ℕ) : Prop := 6 ≤ n
```

In order to do anything with our propositions, we need to use `lemma`s. These allow us to make assertions using type `Prop`, which we then need to prove using a mix of `tactic`s (basic building blocks for arguments) and other `lemma`s. We provide a description of the anatomy of a lemma using the following example:

```
/-- Applies the lemma `nat.not_succ_le_zero` which says for any natural
    number `n`, `n + 1`, cannot be less than or equal to `0`. This uses `5`
    as an argument. -/
lemma not_zero_le_six : ¬ six_le 0 :=
    nat.not_succ_le_zero 5 -- this is the proof
```

Here, the name of the lemma is `not_zero_le_six`. The statement we want to prove, which follows the colon after the name of the lemma, is `¬ six_le 0`. This is defined above as `¬ 6 ≤ 0`. The proof, which follows the `:=` after the statement, uses a lemma from `mathlib`, which states that for any natural number `n`, `n + 1` cannot be less than or equal to `0`. For another example, consider:

```
/-- Applies the lemma `nat.le_succ`, which says for any natural number `
    n`, `n ≤ n + 1` using `6` as the argument -/
lemma six_le_seven : six_le 7 :=
    nat.le_succ 6 -- this is the proof
```

Another consideration we need to make regards set cardinalities. Typically in `lean`, when we want to work with cardinalities of objects, we are required to prove that they are finite (or in some cases, output a junk value[3] of 0 for their cardinality if they are infinite).

[3]Partial functions are difficult to work with in dependent type theory. For this reason, we use "junk values" to avoid this problem. For a treatment of partial functions, see [2, ch. 7].

This is true for types as well as for sets; for the former we have `fintype.card`, and for the latter we have `finset.card` and `ncard`.

```
/- `fintype.card` takes in a type α, with an implicit proof that it is a `
   fintype`, or finite type, and outputs a term of type `ℕ` -/
#check fintype.card
-- output: `fintype.card : Π (α : Type u_2) [_inst_1 : fintype α], ℕ`

/- `finset.card` takes in a `finset`, which is defined as a set with a
   proof of finiteness, and outputs a term of type `ℕ` -/
#check finset.card
-- output: `finset.card : finset ?M_1 → ℕ`

/- `ncard` takes in a `set`, and outputs a term of type `ℕ`. If the set is
   infinite, `ncard` outputs a junk value of `0`. -/
#check set.ncard
-- output: `set.ncard : set ?M_1 → ℕ`
```

However, we wanted to allow for infinite matroids, so we need to be able to work with sets of infinite cardinality. This meant we needed to define a notion of cardinality that allows infinity as an output. The way that we do this is by using `enat`, which extends the natural numbers to include infinity.

```
import mathlib.data.set.ncard.lean

/- This defines `enat` as the union of `ℕ` with an additional element `⊤`,
   an infinity element. -/
def enat : Type := with_top ℕ

notation `ℕ∞` := enat

/- This takes in any `set` and outputs a term of type `ℕ∞`. -/
#check set.encard
-- output: `set.encard : set ?M_1 → ℕ∞`

/- This is equal to infinity. -/
#check (⊤ : enat)
-- output: `⊤ : enat`

/- this produces an error, as there is no maximal element of `ℕ` -/
#check (⊤ : ℕ)
-- output: failed to synthesize type class instance for `has_top ℕ`
```

If `S` is a finite set, is possible to go between `encard S` and the finite cardinality definitions. We have several helper lemmas that do this, which are included in the file `prelim.lean`.

One last type that is important for our formalization is `fin n`, for `n : ℕ`. This is a subtype of ℕ where we only consider elements of ℕ that are less than `n`.

```
/- `fin 2` is a type -/
#check (fin 2)
-- output: `fin 2 : Type`

/- `fintype.card (fin 2) is defined, as `fin` comes with a proof of
   finiteness -/
#check fintype.card (fin 2)
-- output: `fintype.card (fin 2) : ℕ`

/-- The `fintype` cardinality of `fin 7` is `7` -/
lemma card_fin_7 : fintype.card (fin 7) = 7 :=
/- The proof, using lemma `fintype.card_fin`, which takes in `n : ℕ` and
   outputs `fintype.card (fin n) = n` -/
    fintype.card_fin 7 -- this is the proof
```

Trivial proofs can often be written in a single line. However, sometimes we have proofs that can require more steps. For this, we have have a proof mode that we call "tactic mode", which is enclosed in a `begin .. end` block, and allows us to use tactics. An example of a common tactic is rewrite, or `rw`, which allows us to rewrite along equalities.

```
/-- The `finset` cardinality of the `finset` defined as the universe of `
   fin 7`, written `finset.univ`, is `7` -/
lemma finset.card_fin_7 : finset.card (finset.univ : finset (fin 7)) = 7 :
   =
begin
/- rewrite using `finset.card_univ`, which says that for `fintype α`, we
   have `(finset.univ : finset α).card = fintype.card α` -/
  rw finset.card_univ,
/- rewrite using `fintype.card_fin`, which takes in `n : ℕ` and outputs `
   fintype.card (fin n) = n` -/
  rw fintype.card_fin
/- We are done now, as the last rewrite changed the goal to `7 = 7`,
   which the compiler closes for us. -/
end
```

For some less trivial examples, we prove the same thing for `ncard` and `encard`. Here we introduce the `apply` and `refine` tactics as well. The `apply` tactic allows us to take a

lemma of the form `P → Q`, and if the goal is `Q`, we replace it with `P`. The `refine` tactic allows us to divide a goal `P ∧ Q` into two subgoals `P` and `Q`.

```
/-- `ncard` of the `set` defined as the universe of `fin 7`, written `
    set.univ`, is `7` -/
lemma set.ncard_fin_7 : set.ncard (set.univ : set (fin 7)) = 7 :=
begin
/- We can group rewrites together -/
  rw [set.ncard_eq_to_finset_card, set.finite.to_finset_univ,
    finset.card_fin],
end

/-- `ncard` of the `set` defined as the universe of `fin 7`, written `
    set.univ`, is `↑7 : ℕ∞` -/
lemma set.encard_fin_7 : set.encard (set.univ : set (fin 7)) = ↑7 :=
begin
/- Rewrite using `set.encard_eq_coe_iff`, which says that
  `s.encard = ↑k ↔ s.finite ∧ s.ncard = k` -/
  rw set.encard_eq_coe_iff,
/- After the rewrite, we have a goal that looks like `P ∧ Q`, so we can
  use `refine` to split the goal into two subgoals. -/
  refine ⟨_, _⟩,
  { /- Applies `set.finite_univ`, which says that if `α` is a `fintype`,
       then `set.univ α` is finite -/
    apply set.finite_univ },
  { /- Applies our previous lemma, which says that `ncard` of `univ` is 7
    -/
    apply set.ncard_fin_7 },
end
```

Here, `7` has a type coercion from ℕ to ℕ∞, as `encard` is defined as having type ℕ∞. This is denoted as ↑7.

Of course, this is not typically how code is written in mathlib or in this thesis, as we don't need to actually break proofs down into these small pieces. A shorter proof of the last lemma might look like the following:

```
lemma set.encard_fin_7' : set.encard (set.univ : set (fin 7)) = ↑7 :=
  by refine set.encard_eq_coe_iff.2 ⟨set.finite_univ, set.ncard_fin_7⟩
```

2.2 Minors, Extensions & Isomorphism

In this section, we define fundamental operations on matroids. Among them are deletion, contraction, and parallel and series extensions. These concepts are best visualized in the context of graphic matroids. Our definition of a graph permits multiple edges and loops.

Definition 6. *Let $G = (V, E)$ be some graph on vertex set V and edge set E, and let $\mathcal{I}$ denote the collection of acyclic subsets of E. Then $(E, \mathcal{I})$ is a matroid on E, we call it the graphic matroid of G, denoted $M(G)$.*

Then, the collection of bases $\mathcal{B}$ of $M(G)$ is simply the maximal acyclic subsets of E, i.e. the spanning trees or spanning forests of G (where we ignore isolated vertices).

2.2.1 Matroid Minors

First, we discuss deletion.

Definition 7. *Let M be the matroid $(E, \mathcal{I})$ and suppose $X \subseteq E(M)$. Let $\mathcal{I}|X := \{I \subseteq X : I \in \mathcal{I}\}$. Then it is easy to verify that $(X, \mathcal{I}|X)$ is also a matroid. We call this matroid the restriction of M to X, or the deletion of $E - X$ from M, typically denoted $M|X$ or $M\backslash(E - X)$.*

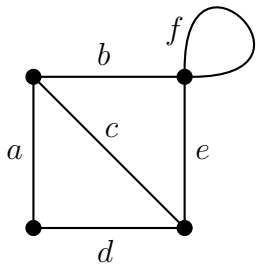
In the context of graphic matroids, we are merely deleting edges of the graph. Let $G = (V, E)$ be the following graph, and suppose we delete edges a, c. Then $M(G\backslash\{a, c\}) = M(G)\backslash\{a, c\}$.

It can be easily verified that the independence axioms hold for the deletion of a matroid, so we will move on to a discussion of dual matroids. We will use the definition of dual matroids that uses bases.

Definition 8. *If $M = (E, \mathcal{B})$ is a matroid and $\mathcal{B}^*(M) := \{E(M) - B : B \in \mathcal{B}(M)\}$, then $(E, \mathcal{B}^*(M))$ is a matroid, and we call it the dual of M.*

Dual matroids are difficult to visualize in general. We can narrow the context of our visualization down to that of graphic matroids of planar graphs, in which the dual matroid is equivalent to taking a planar dual[4] of a fixed planar embedding of the graph. Note that while a planar graph can have multiple, non-isomorphic duals, their graphic matroids will be isomorphic [31]. Let the following be a planar embedding of G:

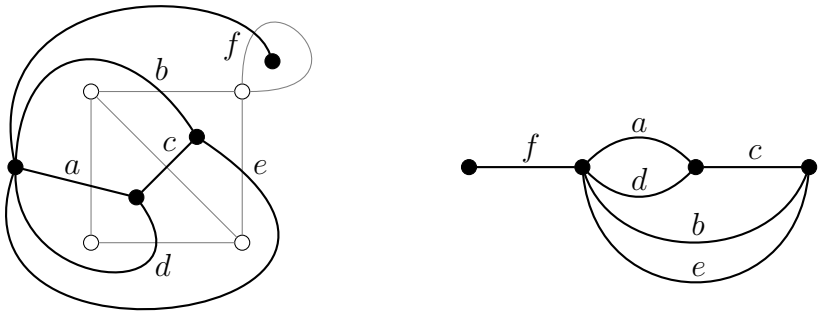
Planar embedding of G



The bases of $M(G)$, or the collection of maximal acyclic edge sets of G, is therefore $\mathcal{B} = \{\{a, b, c\}, \{a, b, d\}, \{a, b, e\}, \{a, c, e\}, \{a, d, e\}, \{b, c, d\}, \{b, d, e\}, \{c, d, e\}\}$. These are spanning trees of G, though G doesn't necessarily need to be connected, in which case $\mathcal{B}$ would be the collection of spanning forests of G.

[4]Planar embeddings of graphs are notoriously difficult to formally define. This is one of the major obstacles to a formalization of the excluded minor characterization for graphic matroids. For examples of work in this area, see [26][35].

Planar dual of the embedding of G



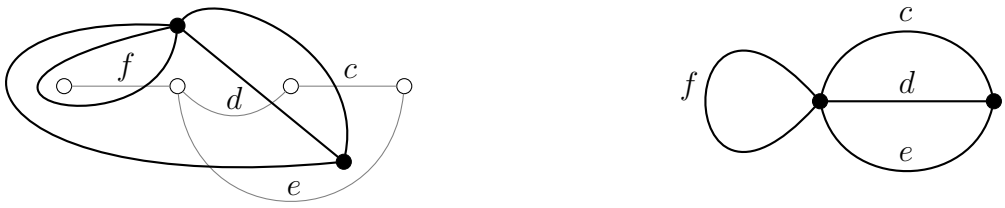
Consider the graph G^* embedded in the plane, dual to our embedding of G, such that the edge set of G^* is equal to that of G. As a reminder, G^* depends on this particular embedding of G, and * is not a function of G. Per our definition of the dual matroid, the collection of bases of $M(G)$ should be $\mathcal{B}^* = \{E - B : B \in \mathcal{B}\}$, so we have $\mathcal{B}^* = \{\{a, b, f\}, \{a, c, f\}, \{a, e, f\}, \{b, c, f\}, \{b, d, f\}, \{c, d, f\}, \{c, e, f\}, \{d, e, f\}\}\}$. Indeed, one can verify that this is the collection of maximal acyclic subsets of the edge set of the dual of the planar embedding, and that $M(G^*) = M(G)^*$.

Lastly, we discuss contraction. Contraction is the dual of deletion:

Definition 9. *If $M = (E, \mathcal{I})$ and $C \subseteq E$, we define contraction of M by C as $M/C := (M^* \backslash C)^*$.*

We can observe that M/C has ground set $M - C$. Furthermore, if C is independent, $I \in \mathcal{I}(M/C)$ if and only if $I \cup C \in \mathcal{I}(M)$. Contracting a subset of E in $M(G)$ is the same as edge contraction in G. Continuing with our same example G, consider the dual of the planar embedding G^* that we derived in the previous example, and suppose we delete edges a, b from G^*. Fix the following planar embedding for G^*:

Now consider the dual of our planar embedding of $G^*\backslash\{a,b\}$.



This graph is "equal" to $G/\{a,b\}$. Note, however, that we cannot say in general that $(G^*\backslash\{a,b\})^* = G/\{a,b\}$. This statement is effectively meaningless, as the dual depends on the particular graph embedding, and * is not a function on G. However, what will be true is $(M(G)^*\backslash C)^* = M(G)/C$, where * is defined as the matroid dual.

Finally, we can define matroid minors.

Definition 10. *If $M = (E, \mathcal{B})$ and $M' = (E', \mathcal{B}')$ are matroids with $E' \subset E$, if M' can be obtained from M by a series of contractions and deletions, we say M' is a minor of M. This is typically denoted $M' \leq M$.*

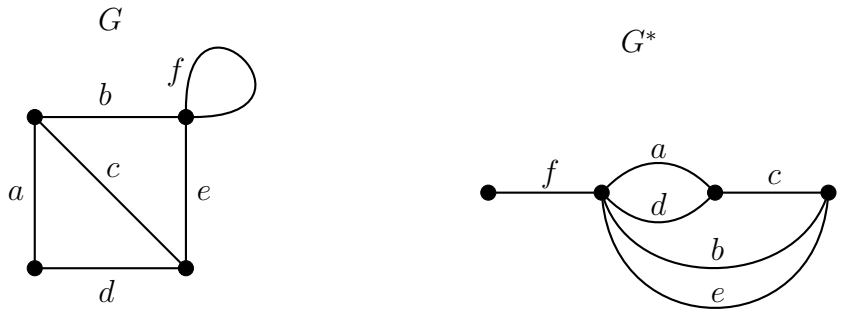
Contraction and deletion commute, so we can equivalently say M' is a minor of M if there exist disjoint subsets $C, D \subseteq E(M)$ such that $M' = M/C\backslash D$.

Definition 11. *If $M = (E, \mathcal{B})$ and $N = (E_N, \mathcal{B}_N)$ are matroids, we say M has an N-minor if there exists some matroid M' such that M' is a minor of M, and N is isomorphic to M'.*

2.2.2 Single Extensions

One portion of this formalization involves showing that matroids minimal with respect to non-representability (more on this later) cannot have loops, coloops, two-element circuits, or two-element cocircuits. We originally started showing this by deleting and contracting elements and then adding them back, but we reconsidered and decided to use a more natural construction for this purpose: parallel extensions and series extensions.

First, we define loops, coloops, circuits, cocircuits, parallel pairs, and series pairs. Consider once again our graph G with fixed planar embedding, and G^* derived by taking the dual of our planar embedding of G, taken from section 2.2.1.



Observe that f is a loop of G. In the graphic matroid $M(G)$, we said that a subset $X \subseteq E$ is independent if and only if it is acyclic in G. However, we can observe that $\{f\}$ by itself forms a one-element cycle in G, so it is a dependent set. Much like in graph theory, we also call one-element dependent sets of matroids loops. We discussed the dual of a matroid in the previous section, so we can also discuss the dual notion of loops.

Definition 12. *An element $e \in E(M)$ is said to be a loop if $\{e\} \notin \mathcal{I}$. In other words, $\{e\}$ is a dependent set.*

Definition 13. *An element $e \in E(M)$ is said to be a coloop if e is a loop of M^*.*

This means that f is a coloop of $M(G)^*$ (as duality of matroids is an involutive operation). Now, an interesting property of coloops is that they must appear in every base of a matroid. And indeed, f appears in every spanning tree of G^*. This makes representations of matroids with loops and coloops fairly easy to work with, as we'll see in later sections. Now, loops are an example of a notion that can also be used to define matroids: minimal dependent sets, which gives us the following definitions:

Definition 14. *Let $X \subseteq E(M)$ be a minimal dependent set of M. In other words, all proper subsets of X are independent, but X is not. Then we say X is a circuit of M.*

Definition 15. *If $X \subseteq E(M)$ is a circuit of M^*, then we say X is a cocircuit of M.*

In our example dual matroid $M(G)^*$, a and d form a circuit (i.e. both $\{a\}$ and $\{d\}$ are independent but $\{a, d\}$ is not). Much like in graph theory, we would call a, d a parallel pair. In $M(G)$, which is the dual of $M(G)^*$, we would call a, d a series pair. This leads us to two more important matroid operations: parallel extension and series extension.

Suppose we have a nonloop edge e of graph G', we can perform a parallel extension on e by adding a new element f to $E(G')$ such that $\{e, f\}$ share the same endpoints and form a cycle in G'.

This operation extends to matroids in a natural way. If M, N are matroids with $f \in E(M)$ such that f is contained in some 2-element circuit $\{e, f\}$ of M, then if $M \backslash f = N$, we say M is a *parallel extension* of e by f in N. If we were to return to our example dual matroid $M(G)^*$, and deleted a, adding it back to $M(G)^* \backslash a$ would be a parallel extension of d by a.

Series extension is the dual of parallel extension. In the context of graphic matroids, it can be thought of as edge subdivision. In the context of graph theory, an edge subdivision typically replaces edge $e = uv$ and edges $e' = uw, e'' = wv$, adding new vertex w in the process. However in the context of graphic matroids, when we perform a series extension on e, we keep e as an object of the new matroid.

We say M^* is a series extension of N^* if and only if M is a parallel extension of N. In other words, M is a *series extension* of e by f in N if $M/f = (M^* \backslash f)^* = N$. In this case, $\{e, f\}$ form a cocircuit in M by duality. In our example matroid $M(G)$, if we were to contract a, we could perform a series extension of d by adding a back to $M(G)/a$ (i.e. subdividing edge d and relabeling to get d and a).

An important concept related to circuits is the fundamental circuit, which we will define via flats and closure:

Definition 16. *If $F \subseteq E$ in a matroid $M = (E, \mathcal{B})$, and I is a basis for F, we say that F is a flat of M if, for all $X \subseteq E$ such that I is a basis for X, we have $X \subseteq F$.*

Definition 17. *If $X \subseteq E$ in matroid $M = (E, \mathcal{B})$, define $cl := \bigcap_{F \supseteq X : F \text{ is a flat of } M} F$. We call this set the closure of X.*

In our graphic matroid example $M(G)$, the closure of $\varnothing$ is $\{f\}$. Closure gives rise to some useful definitions, one of which builds on the circuit definition we provided in the previous section.

Definition 18. *Let $M = (E, \mathcal{I})$ be a matroid defined with respect to its collection of independent sets, and let $x \in E$ and $I \subseteq \mathcal{I}$. If $x \in cl(I)$, then $I \cup \{e\}$ contains a unique circuit $C(e, I)$ such that $e \in C(e, I)$. This is called the fundamental circuit of e with respect to I.*

By convention, if $e \in I$, we say $C(e, I) = \{e\}$. Again returning to $M(G)$, the fundamental circuit $\{c\}$ with respect to $\{a, b, d\}$ is $\{a, d, c\}$.

2.2.3 Lean Implementation

The implementation of matroid minors in Lean is where our decision on whether to treat the ground set of matroids as sets or types becomes important. As mentioned briefly in 2.1, we want to perform operations on matroids (deletion, contraction) that remove elements from their ground sets. If we defined matroids as having ground sets that are types, we would be forced to define a new subtype for each minor. Type equality is a non-starter.

Our definition of matroid restriction is fairly typical: the restriction of M to $X \subseteq E(M)$ is the matroid $M||X$ with ground set X, where for all $I \subseteq X$, we have that $I \in \mathcal{I}(M||X)$ only if $I \in \mathcal{I}(M)$. It is bad practice to write new code if you can reuse something you have already written, so we defined deletion of $D \subseteq E(M)$ from M as the restriction of M to $E(M) - D$.

```
/-- Restrict the matroid M to X : set α. Notation: `M ‖ X`. -/
def restrict (M : matroid α) (X : set α) : matroid α :=
    matroid_of_indep (X ∩ M.E)
    (λ I, M.indep I ∧ I ⊆ X ∩ M.E) ⟨M.empty_indep, empty_subset _⟩
    (λ I J hJ hIJ, ⟨hJ.1.subset hIJ, hIJ.trans hJ.2⟩)
    /- Proofs that the axioms hold -/
    [...]

def delete (M : matroid α) (D : set α) : matroid α := M ‖ Dᶜ
```

While we omit the proofs that the axioms hold, they are only trivial for restrictions of matroids of finite rank [4].

One can define contraction by demonstrating how independent sets are affected, but in our implementation, that would require us to re-prove all of the axioms, so instead we use the fact that matroid contraction is equivalent to the dual of deletion in the dual matroid. i.e. $M/C = (M^* \backslash D)^*$. This allows us to directly produce a new matroid via the definition of dual and the definition of deletion.

```
def contract (M : matroid α) (C : set α) : matroid α := (M* \ C)*
```

A minor of matroid M is a matroid we can derive from M from a series of deletions and contractions of subsets of $E(M)$. We use notation `M' ≤m M` for the following definition.

```
/-- `M` is a minor of `M'`, denoted `M' ≤m M`, if there exist disjoint sets `
    C, D ⊆ M.E` such that `M' = M / C \ D` -/
def minor (M' M : matroid α) : Prop :=
    ∃ (C ⊆ M.E) (D ⊆ M.E), disjoint C D ∧ M' = M / C \ D
```

Then we can define an excluded minor `M` for a property `S` as being minimal with respect to taking minors such that `S` is false for `M`. In other words, `M` is an excluded minor for property `S` if for all proper minors `M'` of `M`, `S M'` holds, but we don't have `S M`. As a consequence, we have the lemma `excluded_minor_iff`, which states that `M` is an excluded minor for `S` if and only if `M ∉ S` and for all `e ∈ M.E`, we have the contraction `M / e ∈ S` and the deletion `M \ e ∈ S`.

```
/-- An excluded minor is a minimal nonelement of S -/
def excluded_minor (S : set (matroid α)) (M : matroid α) :=
    M ∈ minimals (≤m) S^c

lemma excluded_minor_iff (S : set (matroid α)) (hS : minor_closed S) :
    excluded_minor S M ↔ M ∉ S ∧ ∀ e ∈ M.E, M / e ∈ S ∧ M \ e ∈ S :=
   [...]
```

Here, S^c is defined as the absolute complement of `S`, where the universe is the type of elements of `S`.

We explained how a `matroid` is defined as a matroid over some ground set that lives in an ambient type α. If two matroids are in the same type α, it is relatively uncomplicated to define minors, but what happens when two matroids have ground sets that are from different types? One might naively assume that requiring that all matroids live in the same ground type would make this easier, but this would be mathematically unnatural. This would be akin to insisting the ground sets of all matroids must be subsets of the integers. Natural examples exist of matroid structures on different mathematical objects: for instance, edge sets of graphs form graphic matroids, the set of vertices of a graph form matchable sets matroids, sets of vectors form representable matroids, etc.

Our solution to this was to make a second matroid minor definition that uses matroid isomorphisms. For matroids `N, M` that may or may not be in the same type, we say `N ≤ i M` if there exists some minor `M' ≤m M` such that `N` is isomorphic to `M'`. This is similar to definition 11, where we say that M has an N-minor.

```
/-- Matroid isomorphism between matroids `M₁, M₂` is an equivalence
    between their ground sets such that their bases are preserved under
    the mapping. -/
structure iso (M₁ : matroid α₁) (M₂ : matroid α₂) extends equiv M₁.E M₂.E :
    =
  (on_base' : ∀ (B : set M₁.E), M₁.base (coe ″ B) ↔ M₂.base (coe ″ (to_fun
    ″ B)))

/-- We have `N ≤i M` if `M` has an `N`-minor; i.e. `N` is isomorphic to a
    minor of `M` -/
def iso_minor (N : matroid β) (M : matroid α) : Prop :=
    ∃ (M' : matroid α), M' ≤m M ∧ nonempty (N ≃i M')
```

We define loops using closure, which we explained in section section 2.2.2, but for now all we will say is it provides us with a definition equivalent to the singleton dependent set definition. Our implementation of `nonloop` requires that the element in question is a member of the ground set, as it will be trivially true for `e ∉ M.E` that `¬ M.loop e`.

```
/-- A flat is a maximal set having a given basis  -/
def flat (M : matroid α) (F : set α) : Prop :=
(∀ {I X}, M.basis I F → M.basis I X → X ⊆ F) ∧ F ⊆ M.E

/-- The closure of a subset of the ground set is the intersection of the
    flats containing it. A set `X` that doesn't satisfy `X ⊆ M.E` has the
    junk value `M.cl X := M.cl (X ∩ M.E)`. -/
def cl (M : matroid α) (X : set α) : set α :=
    ⋂₀ {F | M.flat F ∧ X ∩ M.E ⊆ F}

/-- A loop is a member of the closure of the empty set -/
def loop (M : matroid α) (e : α) : Prop := e ∈ M.cl ∅

/-- A `nonloop` is an element that is not a loop -/
def nonloop (M : matroid α) (e : α) : Prop := ¬ M.loop e ∧ e ∈ M.E

/-- A coloop is a loop of the dual  -/
def coloop (M : matroid α) (e : α) : Prop := M*.loop e

/-- A circuit is a minimal dependent set -/
def circuit (M : matroid α) (C : set α) : Prop :=
    C ∈ minimals (⊆) {X | M.dep X}

/-- A cocircuit is the complement of a hyperplane -/
```

```
def cocircuit (M : matroid α) (K : set α) : Prop := M✶.circuit K

/-- For an independent set `I` that spans a point `e ∉ I`, the unique
    circuit contained in `I ∪ {e}`. Has the junk value `{e}` if `e ∈ I` and `
    univ` if `e ∉ M.cl I`. -/
def fund_circuit (M : matroid α) (e : α) (I : set α) :=
    insert e (⋂₀ {J | J ⊆ I ∧ e ∈ M.cl J})
```

We also list our single extension definitions.

```
/-- add loop element `f` to `M`. -/
def add_loop (M : matroid α) (f : α) : matroid α :=
    M.restrict' (insert f M.E)

/-- add loop element `f` to `M✶` and take the dual, effectively adding a
    coloop to `M`. -/
def add_coloop (M : matroid α) (f : α) : matroid α :=
    (M✶.add_loop f)✶

/-- extend `e` in `M` by a parallel element `f`. -/
def parallel_extend (M : matroid α) (e f : α) : matroid α := M.preimage
    ((@id α).update f e)

/-- extend `e` in `M` by an element `f` in series. -/
def series_extend (M : matroid α) (e f : α) : matroid α :=
    (M✶.parallel_extend e f)✶
```

2.3 Rank & Uniform Matroids

Lastly, for this proof we needed to define uniform matroids, as the excluded minor for binary matroids is a uniform matroid. In order to define uniform matroids, we need a notion of rank.

Definition 19. *The rank of a matroid $M = (E, \mathcal{I})$ is defined as the size of its maximal independent set, or alternatively, the size of a base. This is typically denoted $r(M)$.*

A straightforward consequence of the base axioms is that in a finite-rank matroid, every base of a matroid will have the same cardinality. The same is true for every basis of any subset X of the ground set.

The definition of rank gives us an important class of matroids: Uniform matroids.

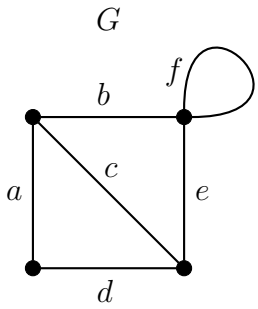


Figure 2.1: If $M(G)$ is the graphic matroid of G, its rank is 3. Every maximal acyclic subset will have size 3.

Definition 20. *If E is a set with cardinality $|E| = n$, we say the uniform matroid $U_{k,n}$ on ground set E is a matroid with rank k, where every subset of E that has cardinality k is a base of $U_{k,n}$.*

If $k > n$, then the uniform matroid $U_{k,n}$ is undefined. A direct consequence of this definition, and the definition of matroid minor, is that if $n', n, k', k \in \mathbb{N}$ such that $n' \leq n$, $k' \leq k$, $k' \leq n'$ and $k \leq n$, then $U_{k',n'}$ is a minor of $U_{k,n}$.

2.3.1 Lean Implementation

Because we wanted to allow for infinite matroids and therefore sets with infinite cardinality, we used `encard`, which is described in section 2.1.2). We first set up a definition of the rank `er` defined on a set `set α` that allows for infinite bases. Using this, we define a finite definition `r` of a `set α` that lives in type $\mathbb{N}$. Finally, we use `r` to define `rk`, the rank of matroid `M`, by taking `M.r M.E`, or the rank of the ground set.

```
/-- The rank `er X` of a set `X` is the cardinality of a basis of `X`,
    allowing for infinite cardinalities -/
def er {α : Type*} (M : matroid α) (X : set α) : ℕ∞ :=
    ⨅ (I : {I | M.basis I (X ∩ M.E)}), encard (I : set α)

/-- The rank function. Intended to be used in a `finite_rk` matroid;
    otherwise `er` is better.-/
def r (M : matroid α) (X : set α) : ℕ := (M.er X).to_nat
```

```
/-- The rank of the ground set of a matroid -/
@[reducible] def rk (M : matroid α) : ℕ := M.r M.E
```

Our formalized definition of closure uses flats, which is outside the scope of this thesis, but we provide their definition here anyway. This formal definition of closure is equivalent to the definition we provided above: $cl(X) = \{x : r(X \cup \{x\}) = r(X)\}$.

Our formalization for uniform matroids uses two different definitions. The first one is defined as having an arbitrary type for the ground set, and the second uses `fin n` as the ground set type. We used `fin n` because it allows us to easily talk about the elements, but `fin n` is finite, and we want to be able to include infinite matroids. However, for the purposes of this thesis, the excluded minor for binary representation is a finite uniform matroid, so we use the `fin n` definition.

```
/-- A uniform matroid with a given rank and ground set -/
def set.unif_on (E : set α) (k : ℕ) := (free_on E).truncate k

/-- A uniform matroid of a given rank whose ground set is the universe
   of a type -/
def unif_on (α : Type*) (k : ℕ) := (univ : set α).unif_on k

/-- A canonical uniform matroid, with rank `a` and ground type `fin b`. -/
def unif (a b : ℕ) := unif_on (fin b) a
```

We define `free_on` as the free matroid on `E`, and `truncate` as the truncation operation of a matroid. Both are outside of the scope of this thesis, but they produce the correct notion of uniform matroids in this definition.

2.4 Representation Definition

If A is a matrix over some field $\mathbb{F}$ with columns indexed by set E, we can define a matroid on E where a set $I \subseteq E$ is independent if and only if its corresponding column vectors in A are linearly independent over $\mathbb{F}$. This is called the column matroid $M(A)$. Then, a matroid M is representable over $\mathbb{F}$ if there is some matrix A with columns indexed by E, where $M = M(A)$. Now, this definition of a matroid representation poses a number of implementation challenges. For one, showing that a matroid is representable would require providing some kind of explicit matrix, or showing that such a matrix exists, and matrices are difficult to work with in the existing `mathlib` API. Furthermore, it is a known fact in matroid theory that row-equivalent matrices represent the same matroid [25].

$$\begin{array}{c} \begin{matrix} a & b & c & d & e & f \end{matrix} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 2 \\ 0 & 1 & 2 & 1 & 2 & 0 \\ 2 & 1 & 1 & 1 & 0 & 1 \end{pmatrix} \end{array} \underset{row\ reduce}{\longrightarrow} \begin{array}{c} \begin{matrix} a & b & c & d & e & f \end{matrix} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 2 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

Figure 2.2: If $M = (E, \mathcal{B})$ with ground set $E = \{a, b, c, d, e, f\}$ and collection of bases $\mathcal{B} = \{\{a, b, c\}, \{a, b, e\}, \{a, c, d\}, \{a, d, e\}, \{b, c, e\}, \{b, c, f\}, \{b, e, f\}, \{c, d, e\}, \{c, d, f\}, \{d, e, f\}\}$, the two matrices above are equivalent representations of M over $GF(3)$.

Unfortunately, there is no API for working with row operations of matrices in mathlib, and it would have been extremely time-consuming to develop our own solely for this purpose. For this reason, we had to experiment with different definitions before we settled on one that worked. Our initial approach to defining the representation of a matroid involved two different definitions: The first being a function from $E(M)$ to a vector space W, and the second being an explicit mapping from $E(M)$ to the vector space $\alpha \to \mathbb{F}$, which more closely resembles the matrix definition of representation outlined above. The arbitrary module definition affords us more flexibility and abstraction, while the indexed vector space $\alpha \to \mathbb{F}$ allows us to more easily define elements of the representation.

Because our definition of a matroid lives in an ambient type α, and our ground set is of type `set α`, we now had two choices for how to define our mapping: either we restrict the mapping to just elements of the ground set and leave it undefined for the rest of α, or we define the mapping over all terms of type α and treat nonelements of the ground set as loops in our definition of representation. While the first definition is more similar to what a mathematician might expect, the problem is that instead of being able to talk about terms and sets of type α, we would be forced to treat the ground set as a type. Then we would have to show that every term is an element of the ground set, and every set we discuss is a subset of the ground set. This makes for a much more tedious implementation.

So, ultimately, our main definition of representation became:

```
structure rep (𝔽 W : Type*) [field 𝔽] [add_comm_group W] [module 𝔽 W]
(M : matroid α) :=
    /- This is the function of the representation -/
    (to_fun : α → W)
    /- This is the condition that a subset of M.E is independent if and
    only if its image in W is linearly independent. -/
    (valid' : ∀ (I ⊆ M.E),
        linear_independent 𝔽 (to_fun ∘ coe : I → W) ↔ M.indep I)
```

```
    /- This is the condition that non-elements of M.E map to 0 in W. -/
    (support : ∀ (e : α), e ∉ M.E → to_fun e = 0)
```

Here, the `support` field explicitly requires that a representation φ maps non-elements of `M.E` to the zero vector in our module. With a modification of the `valid′` property preceding it, we could write a definition of `rep` that doesn't require an explicit condition on non-elements of `M.E`. If we had written `valid′` as a condition on `set α` instead of on subsets of `M.E`, we could have derived `support` as a consequence. However, we decided to keep the two conditions separate for clarity.

Now, as mentioned above, matrix representations of a matroid are equivalent under row operations. For this reason, if a matroid M is representable over a field $\mathbb{F}$, we can always find a representation where, given a base B of M, the columns of the representation corresponding to B form the identity matrix. Our second definition of representation that mapped into the vector space $\alpha \to \mathbb{F}$ eventually evolved into a "standard representation," in which a base B of M serves as our indexing set. This definition invokes our first definition; because we have a representation in arbitrary module `W`, we can define a representation in $B \to \mathbb{F}$.

However, we used a construction more specific than $B \to \mathbb{F}$ for our standard representation. We used $B \to_0 \mathbb{F}$, or `finsupp`, as it is called in Lean, which is the type of mappings from B to $\mathbb{F}$ such that all but a finite number of elements of B map to zero. For our purposes, because we are proving statements about the representability of a matroid of finite rank and B is always finite, $E(M) \to B \to_0 \mathbb{F}$ can be read in more standard mathematical notification as $E(M) \to \mathbb{F}^B$. The following is our standard representation definition:

```
/-- If `φ'` is a representation of matroid `M` and `B` is a base of `M`, we
    derive a standard representation of `M` with respect to `φ'` and `B`
    such that elements of `B` map to columns of the identity matrix and
    other elements of `M.E` are written as linear combinations of elements
    in the image of `B` -/
def standard_rep (φ' : rep 𝔽 W M) {B : set α} (hB : M.base B) :
rep 𝔽 (B →₀ 𝔽) M := {
    to_fun := λ e : α, ((valid φ').2 hB.indep).repr ⟨φ' e, [...]⟩,
    valid' := [...],
    support := [...] }
```

where the first `[...]` is a placeholder for a proof that φ' `e` is contained in the span of the image of φ' of `B`. This definition uses `linear_independent.repr`, which takes in a linearly independent set of vectors and a proof that φ' `e` is in their span, and produces φ' `e` as a linear combination of the independent vectors. Because our representation essentially

consists of constructing a mapping from elements of $E(M)$ to $B \to_0 \mathbb{F}$, we have to provide a proof that the element maps to something spanned by the image of B in our module W. This proof is trivial, but has been omitted for readability.

We also had to define a notion of representability. Now, vector spaces are defined in `lean` as types with additional properties. A vector space is a module where the scalar ring is a field, and the definitions in `mathlib` are written with the intention of maximizing generality, so the following definition taken from [22] is about modules:

```
/-- A module is a generalization of vector spaces to a scalar semiring.
    It consists of a scalar semiring `R` and an additive monoid of
    "vectors" `M`, connected by a "scalar multiplication" operation `r • x :
     M` (where `r : R` and `x : M`) with some natural associativity and
    distributivity axioms similar to those on a ring. -/
@[ext, class]
structure module (R : Type u) (M : Type v) [semiring R]
[add_comm_monoid M] : Type (max u v)
to_distrib_mul_action : distrib_mul_action R M
add_smul : ∀ (r s : R) (x : M), (r + s) • x = r • x + s • x
zero_smul : ∀ (x : M), 0 • x = 0
```

So a `module` is defined as a structure over types `R` and `M` where `R` is a semiring, `M` is an additive commutative monoid, and elements from `R` and `M` have the properties we expect in a module or vector space. However, this is problematic for us. Our first definition is a mapping to an arbitrary vector space W over $\mathbb{F}$, so we had to define representability as the existence of some W and mapping $\varphi : E(M) \to W$ that is a representation of M. There is a danger in quantifying over types, and this definition of representability essentially starts with `∃ (W : Type)`, so we eventually had to modify our definition.

Our current definition of representability is: A matroid M is representable if there exists a base B of M and a function $\varphi : E(M) \to B \to_0 \mathbb{F}$ such that φ is a representation of M. This allows us to still be able to specify an arbitrary module for which M has a representation in, while also deriving a specific module for the definition of representability.

```
/-- A matroid `M` is representable if there exists a base `B` of `M` and a
    representation of `M` in `B →₀ 𝔽` -/
def is_representable (𝔽 : Type*) [field 𝔽] (M : matroid α) : Prop :=
  ∃ (B : set α) (hB : M.base B), nonempty (rep 𝔽 (B →₀ 𝔽) M)
```

Now if we have some representation φ of M, we can derive `is_representable M` using the following lemma and our definition of `standard_rep`.

```
/-- A representation over *any* module certifies representability-/
```

```
lemma is_representable_of_rep {W : Type*} [add_comm_group W] [module 𝔽 W]
    (φ : rep 𝔽 W M) :
     is_representable 𝔽 M :=
  begin
/- Obtain a base of `M` -/
     obtain ⟨B, hB⟩ := M.exists_base,
/- Use `B` and `standard_rep φ hB` to get a representation in `B →₀ 𝔽` -/
     exact ⟨B, hB, ⟨standard_rep φ hB⟩⟩,
  end
```

We have a second way of demonstrating $\mathbb{F}$-representability of a matroid in the form of `matroid_of_module_fun`, which defines a matroid out of any function to a vector space. This generalizes the concept of defining a matroid out of the columns of a matrix, as the columns of a matrix can be thought of as a map from an indexing set (the indices of the columns) to the column vectors themselves.

```
/-- If `v : α → W` is a map from indexing type `α` to elements of a
    module `W` over a field `𝔽`, and `ground : set α`, we can define a
    matroid on the elements of `ground` where independence of subsets `I ⊆
    ground` corresponds to linear independence in `W` -/
def matroid_of_module_fun (𝔽 W : Type*) {α : Type*} [field 𝔽]
    [add_comm_group W] [module 𝔽 W] [finite_dimensional 𝔽 W]
     (v : α → W) (ground : set α) : matroid α := matroid_of_indep_of_bdd'
    ground
     (λ (I : set α), (linear_independent 𝔽 (λ x : I, v x)) ∧ I ⊆ ground)
```

This definition uses `matroid_of_indep_of_bdd'` (defined in section 2.1.3) to construct a matroid with the independence axioms, where sets in this matroid are mapped by `v` to linearly independent sets in the module.

When we define a matroid in this way, we easily get that the function `v` that we pass into `matroid_of_module_fun` is a representation of the matroid we have constructed, and if we can demonstrate that our target matroid `M` is equal to `matroid_of_module_fun`, we can also derive a representation of `M`. This way of proving representability has the advantage of allowing us to use any criterion for matroid equality. As we will see in section 2.6, this gives us a lot of flexibility when defining representations of matroid extensions.

2.4.1 Representations of Uniform Matroids

In `lean`, the way we refer to $GF(2)$ is `zmod 2`.

```
/- A matroid is binary if it has a `GF(2)`-representation -/
def matroid.is_binary (M : matroid α) := M.is_representable (zmod 2)
```

Our first milestone in this project was the formalization of the proof that $U_{2,4}$ is not binary, i.e. not representable in $\mathbb{F}_2$. The proof for this is a straightforward counting argument; roughly: Because $U_{2,4}$ has rank 2, if it is binary, we must have some representation $\varphi : E(U_{2,4}) \to \mathbb{F}_2^2$. Because $U_{2,4}$ is simple, φ must be an injective map to nonzero vectors in $\mathbb{F}_2^2$. However, there are three nonzero vectors in $\mathbb{F}_2^2$, and four distinct non-loop elements of $U_{2,4}$, so we can't have an injective map to the nonzero elements.

```
lemma U24_nonbinary : ¬ matroid.is_binary (unif 2 4) :=
begin
/- Assume for contradiction that `(unif 2 4)` is representable over `zmod
    2` -/
  by_contra hrep,
/- Obtain the representation `φ'` and a `base B` of `(unif 2 4)` -/
  obtain ⟨B, ⟨hB, ⟨φ'⟩⟩⟩ := hrep,
/- The span of the image of `φ'` on the ground set of `unif 2 4` is a
    submodule of `B →₀ zmod 2` -/
  have hsubmodule := @span_mono (zmod 2) _ _ _ _ _ _ (subset_univ (φ' ''
    (unif 2 4).E)),
  [...]
/- The rank of `B →₀ zmod 2` is 2 -/
  have hrank : (finrank (zmod 2) (B →₀ zmod 2)) = 2,
  [...]
/- `fin 2` is a basis for `B →₀ zmod 2` -/
  have hbasis := finite_dimensional.fin_basis (zmod 2) (B →₀ zmod 2),
  [...]
/- `B →₀ zmod 2` has cardinality 4 -/
  have hcard := @module.card_fintype _ (zmod 2) (B →₀ zmod 2) _ _ _ _
    hbasis _ _,
  [...]
/- Because `unif 2 4` is simple, `φ'` must map its elements to nonzero
    elements of `B →₀ zmod 2`. So the cardinality of the image of `φ'` on
    the ground set of `unif 2 4` must be at most the cardinality of `B →₀
    zmod 2` minus the zero vector, which is 3 -/
  have hcardimagele3 :=
-- this part takes an embedding and says the cardinality of the set
    being embedded is at most the cardinality of the target set
  fintype.card_le_of_embedding (embedding_of_subset _ _ (subset_trans
-- this part says `φ'` maps to nonzero elements
  (φ'.subset_nonzero_of_simple (unif_simple 2 4 rfl.le))
```

```
-- this part says the image of `φ'` is a subset of `B →₀ zmod 2` remove 0
  (@diff_subset_diff_left _ _ _ ({0} : set (B →₀ zmod 2)) (span_le.1
   hsubmodule)))),
  [...]
/- The cardinality of the image of `φ'` of the ground set of `unif 2 4` is
   4 -/
  have hcard4 : fintype.card (φ' '' (unif 2 4).E) = fintype.card (fin 4),
  [...]
/- `linarith` produces a contradiction from `4 ≤ 3` -/
  linarith,
  [...]
end
```

Additionally, we also proved that $U_{2,3}$ and $U_{1,3}$ are binary. On paper, these proofs are fairly easy. We present the following binary representations of $U_{2,3}$ and $U_{1,3}$, respectively:

$$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} \qquad\qquad (1 \quad 1 \quad 1)$$

Unfortunately, this was more tedious than we would have liked in our formalization. We omit the more tedious aspects in the following code snippets and focus on the important parts. For the representations of $U_{2,3}$ and $U_{1,3}$, we use the `rep.mk` constructor that allows us to define an object of type `rep 𝔽 W M` by providing a mapping from elements of `M` to module `W` over $\mathbb{F}$, as well as proofs that the mapping respects independence of sets and maps non-elements of `M.E` to the zero vector in `W`.

```
/-- `unif 2 3` is representable over `zmod 2`. -/
lemma U23_binary : matroid.is_binary (unif 2 3) :=
begin
/- The cardinality of `(@set.univ (fin 2 → zmod 2)) \ 0` is 3 -/
  have hcard3 : fintype.card ((set.univ \ {0}) : set (fin 2 → zmod 2)) =
   3,
  [...]
/- We have an equivalence relation between the ground set of `unif 2 3`
   and `(@set.univ (fin 2 → zmod 2)) \ 0` -/
  have f := equiv.symm (fintype.equiv_fin_of_card_eq hcard3),
/- Define our representation as having `f` as our `to_fun`, with a short
   proof that `support` holds -/
  have φ := @rep.mk _ (zmod 2) (fin 2 → zmod 2) _ _ _ (unif 2 3) (λ x,
   (f x)) (λ I hI, _) ([...]),
```

```
  { /- close the main goal by using the representation -/
    rw [matroid.is_binary, is_representable],
    apply is_representable_of_rep φ },
/- Modify the goal so that we are proving that the image of `φ` on a
   subset of the ground set is linearly independent if and only if its
   size is at most 2 -/
    rw [unif_indep_iff],
/- Now we prove that φ respects independence. We omit the rest of this
   proof, as on paper it is simple but the formal proof involves some
   tedious set cardinality arguments. -/
  [...]
end
```

We proved that $U_{1,3}$ is binary by proving the more general statement, that $U_{1,k}$ for arbitrary $k \geq 1$ is representable over any field $\mathbb{F}$. This was fairly straightforward, since we can easily construct a representation for $U_{1,k}$ over $\mathbb{F}$, viewed as a one-dimensional module over the field $\mathbb{F}$. We omit the proof because it is similar to the proof for `U23_binary`, and instead provide only the statement.

```
/-- For `k : ℕ` such that `1 ≤ k`, `unif 1 k` is representable over all `
   nontrivial 𝔽`. -/
lemma U1k_representable (k : ℕ) (hk : 1 ≤ k) [nontrivial 𝔽] : (unif 1
   k).is_representable 𝔽 := [...]
```

Together, these lemmas give us the following lemma.

```
/-- `unif 2 4` is an excluded minor for binary representation -/
lemma U24_excluded_minor : excluded_minor (set_of matroid.is_binary)
   (unif 2 4) :=
begin
/- Modify the goal so that we have to prove that `unif 2 4` is not
   binary, and its minors are -/
  apply (excluded_minor_iff (set_of matroid.is_binary) (@minor_closed_rep
    _ (zmod 2) _)).2
    ⟨U24_nonbinary, λ e he, ⟨_, _⟩⟩, -- `λ e he` selects some element of `
   unif 2 4`
  { /- Obtain our representation of `unif 1 3` -/
    obtain ⟨B, ⟨hB, ⟨φc⟩⟩⟩ := @U1k_representable (zmod 2) _ 3 _ _,
    /- Obtain an isomorphism `ψc` between `unif 2 4 / e` and `unif 1 3` -/
    obtain ⟨ψc⟩ := (contract_elem_unif 1 3 e),
    /- Construct our representation of `unif 2 4 / e` -/
    apply is_representable_of_rep (rep_of_iso _ _ ψc φc),
    [...] },
```

```
  { /- Obtain our representation of `unif 2 3` -/
    obtain ⟨B, ⟨hB, ⟨φc⟩⟩⟩ := @U23_binary,
    /- Obtain an isomorphism `ψd` between `unif 2 4 \ e` and `unif 2 3` -/
    obtain ⟨ψd⟩ := (delete_elem_unif 2 3 e),
    /- Construct our representation of `unif 2 4 \ e` -/
    apply is_representable_of_rep (rep_of_iso _ _ ψd φc) },
end
```

Along the way of formalizing these facts, we built a rudimentary API for reasoning about one-dimensional subspaces of a vector space. This approach was intended for us to be able to show arbitrary $U_{2,k}$ representability. At the time that we were developing this proof, there was no API built for reasoning about k-dimensional subspaces of a vector space (i.e. Grassmannians), so we had to count the one-dimensional subspaces directly. Our formalization of the number of one-dimensional subspaces, as well as counting arbitrary k-dimensional subspaces, is something that we intend to contribute to mathlib.

There is a way to prove $\mathbb{F}$-representability for arbitrary $U_{k,n}$ that involves a different definition of representation as well as Vandermonde matrices that has been completed in Lean 4, but it is outside the scope of this thesis.

2.5 Representability of Matroid Minors

The purpose of this section is to show how we implemented the fact that if a matroid is $\mathbb{F}$-representable, so are all of its minors.

2.5.1 Deletion

If we have a matroid M with $e \in E(M)$ represented by matrix A, we can derive a representation of $M\backslash e$ by simply deleting the column of A corresponding to e. For instance, suppose that we have matroid M with $E(M) = \{a, b, c, d, e, f\}$ that is represented by

$$A = \begin{array}{c} \begin{matrix} a & b & c & d & e & f \end{matrix} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 2 \\ 0 & 1 & 2 & 1 & 2 & 0 \\ 2 & 0 & 1 & 1 & 0 & 1 \end{pmatrix} \end{array}$$

Then suppose we want to delete b, c, and f from M. Then the following matroid will represent $M\backslash\{b, c, f\}$, which is fairly straightforward to verify:

$$A' = \begin{array}{c} \begin{matrix} a & d & e \end{matrix} \\ \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 2 \\ 2 & 1 & 0 \end{pmatrix} \end{array}$$

We do not use matrices for our representation definition, but the corresponding operation was relatively easy to define in Lean. Instead of deleting a column of a matrix, we simply restrict the domain of our representation function to $E(M)\backslash D$, and map elements of D to 0 (building on our `support` condition for representations that requires non-elements to map to 0). We define this with a simple `if-then-else` function as follows:

```
/-- If `M` has a representation `φ` over 𝔽, and `D : set α`, we can
    construct a representation of `M \ D`-/
def rep_of_del (M : matroid α) (φ : rep 𝔽 W N) (D : set α) :
rep 𝔽 W (M \ D) :=
    /- This function maps elements of `D` to `0`, and otherwise maps `x : α`
     to `φ x` -/
    to_fun := λ x, if x ∈ D then 0 else φ x,
    valid' := [...],
    support := [...] /- This is very easy to prove, as elements outside
    of `M.E` already map to `0`, and now elements of D map to `0` as well. -/
```

This, together with a proof that this function preserves independence of sets, and maps non-elements of $E(M)$ to zero, gives us an object of type `rep 𝔽 W (N \ D)`, as desired.

2.5.2 Contraction

We now wish to prove that contracting a set in a representable matroid yields a representable matroid. One way to do this mathematically is by first proving the the dual of a representable matroid is representable, and then using the fact that contraction is dual to deletion. However, we had some difficulty defining the representation of the dual of a matroid.

The most principled way to define the representation of a dual is using orthogonal vector spaces. While orthogonal vector spaces have been defined in Lean 3 `mathlib` as

`dual_annihilator`, this definition does not have the properties that we want for representations. It is in a sense too general, which is unwieldy for our purposes. This meant that we could not directly define representations of dual matroids. Due to this difficulty, we decided to go for a more direct route and define the representation of a minor contraction explicitly.

If A is a matrix representation of matroid M and $y \in E(M)$, when y is not a loop of M, contracting it is equivalent to pivoting on a non-zero entry of its corresponding column, deleting that row from A, and then deleting the column from A (pg 108 in Oxley). For example, suppose that we have matroid M with $E(M) = \{a, b, c, d, e, f\}$ which is represented by

$$\begin{matrix} a & b & c & d & e & f \end{matrix}$$
$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 2 \\ 0 & 1 & 2 & 1 & 2 & 0 \\ 2 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

and suppose we want to contract e. First we find a nonzero entry of the column of A corresponding to e; let us pick the first 1 entry. We delete that row from the matrix to get

$$\begin{matrix} a & b & c & d & e & f \end{matrix}$$
$$\begin{pmatrix} 0 & 1 & 2 & 1 & 2 & 0 \\ 2 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

Now we delete the column corresponding to e to get

$$\begin{matrix} a & b & c & d & f \end{matrix}$$
$$\begin{pmatrix} 0 & 1 & 2 & 1 & 0 \\ 2 & 0 & 1 & 1 & 1 \end{pmatrix}$$

This matrix is a representation of M/e.

However, this method of constructing a representation of a matroid with a contracted element is a hindrance to formalizing the definition. As explained in Section 2.2, in our discussion about the definition of a representation, matrices are problematic. For contraction specifically, using a matrix would require us to invoke the axiom of choice to pick a

row for which the column has a nonzero entry in order to delete it, as there is no concept of a "first" entry. This operation drastically changes the representation, and we would have to modify the matrix itself.

So this leaves us with the question: How can we avoid using matrices in the context of matroid contractions, especially since we also cannot define it as the representation of the dual of a deletion? To answer that question, we make the observation that deleting a row and column of a matrix is the same as projecting into a vector space of dimension one smaller. Luckily for us, Lean 3 `mathlib` has a submodule quotient API that allows us to do exactly that. Our definition of the representation of a matroid with a contracted set looks roughly like:

```
/-- If `M` has a representation `φ` over 𝔽, and `C : set α` is a subset of `
   M.E`, we can construct a representation of `M / C`-/
def rep_of_contr (M : matroid α) (φ : rep 𝔽 W M) (C : set α)
(hC : C ⊆ M.E): rep 𝔽 (W / span 𝔽 (φ '' C)) (M / C) := {
    /- Maps elements of M to the projection of their image under φ to
    the smaller vector space -/
    to_fun := λ x, submodule.quotient.mk (φ x),
    valid' := [...],
    support := [...] }
```

where the function `submodule.quotient.mk` is defined as the map associating to an element of module W the corresponding element of W/S, when S is a submodule of W. The type of our resultant object is a representation of M/C over the module obtained by quotienting W by the span of the image of C under φ. This is where our first definition of representations in an arbitrary module gets to shine - no fiddling with rows and columns, just straightforward[5] reasoning about linear independence of vectors in the projection to a module of smaller dimension.

2.5.3 Representability is a Minor-Closed Class

With the work described in this section, and some careful definitions, the end result of this section is a simple lemma with a short proof:

```
/-- A minor of an 𝔽-representable matroid is 𝔽-representable -/
def is_rep_of_minor_of_is_rep (N : matroid α) (hNM : N ≤m M)
```

[5]In Lean 3 `mathlib`, there are no lemmas that talk about linear independence in the projection of a module, so we had to use lemmas about the preservation of linear independence along linear maps and reason about independence of the union of sets. This is still preferable to using matrices.

```
(hM : M.is_representable 𝔽) : N.is_representable 𝔽 :=
begin
/- Obtain base `B` and representation `φ` -/
  obtain ⟨B, ⟨hB, ⟨φ⟩⟩⟩ := hM,
/- Since `N` is a minor of `M`, there exist `C, D ⊆ M.E` that we contract
   and delete respectively in `M` to get `N` -/
  obtain ⟨C, ⟨D, ⟨hC, ⟨hD, ⟨hCD, rfl⟩⟩⟩⟩⟩ :=
    minor.exists_contract_indep_delete_coindep hNM,
/- Apply our defitions `rep_of_contr` for contracting `C` and `rep_of_del`
   for deleting `D` to construct representation of `N` -/
  apply is_representable_of_rep (rep_of_del (M / C)
    (rep_of_contr M φ C hC.subset_ground) D),
end
```

i.e. if N is a minor of M and M is representable over $\mathbb{F}$, then so is N. This gives us another conclusion that's useful for talking about excluded minors for representability:

```
/-- Representability is a minor-closed class -/
lemma minor_closed_rep :
minor_closed (matroid.is_representable 𝔽 : matroid α → Prop) :=
    λ M N hNM hM, is_rep_of_minor_of_is_rep N hNM hM
```

which directly plugs in `is_rep_of_minor_of_is_rep`. In other words, $\mathbb{F}$-representability is a minor-closed class of matroids.

2.6 Representability of Matroid Extensions

In the previous section, we discussed matroids that have representations, and the way that those representations change for their minors. However, we now needed a way to go in the opposite direction: deriving a representation of a matroid from a representation of a minor, given that the rest of the matroid followed certain properties. This is where parallel and series extensions come in.

In the proof of the excluded minor characterization of binary matroids, we can rule out matroids that contain 1- or 2- element circuits and cocircuits. This is because, if a matroid is an excluded minor for representability, but it contains a 1- or 2- element circuit, we can easily delete an element to find a representation of its minor, and extend it to a representation of the whole matroid. We can also rule out 1- or 2- element cocircuits by duality.

2.6.1 Loops and Coloops

If a matroid M is an excluded minor for representability over field $\mathbb{F}$, it is pretty trivial to show that it cannot contain loops: Suppose for contradiction that $e \in E(M)$ is a loop. Then, since M is an excluded minor for representability, the matroid $M\backslash e$ is representable over $\mathbb{F}$, so let φ be a representation of $M\backslash e$. Then we can extend φ to a representation of M by mapping e to 0.

$$\begin{array}{c} \begin{matrix} a & b & c & d & e & f \end{matrix} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 2 \\ 0 & 1 & 2 & 1 & 2 & 0 \\ 2 & 0 & 1 & 1 & 0 & 1 \end{pmatrix} \end{array} \xrightarrow{\textit{adding loop } x} \begin{array}{c} \begin{matrix} a & b & c & d & e & f & x \end{matrix} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 2 & 0 \\ 0 & 1 & 2 & 1 & 2 & 0 & 0 \\ 2 & 0 & 1 & 1 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

Figure 2.3: If we have a matrix representation A of $M\backslash x$, and x is a loop of M, we can extend A to a representation of M by adding a column of all zeroes for x.

Our `lean` implementation needs no modification, as a representation φ `: rep 𝔽 W M` maps loops and non-elements of `M.E` to the same image: `0`. This is a convenient consequence of our definition of representation requiring non-elements to map to `0`.

```
/-- If `f : α` is a loop element of `M.E` and `M \ f` has a representation
    over 𝔽, we can construct a representation of `M` -/
def rep_of_loop (M : matroid α) [finite_rk M] {f : α} (hf : M.loop f)
  (φ : rep 𝔽 W (M \ f)) : rep 𝔽 W M := {
    /- Since φ already maps non-elements of (M \ f).E to 0, we have φ f
    = 0, which is what we want. -/
    to_fun := φ,
    valid' := [...],
    support := [...] }
```

The situation is trickier for adding a coloop, as a coloop of a matroid must be contained in every base. Extending a matrix representation by adding a coloop requires us to add a new row in addition to the new column, such that the new row and column have all zero entries except for the entry corresponding to the coloop.

For our `add_coloop` representation, we constructed a new module $(W \times \mathbb{F})$ to represent the series extension of M, and defined a piecewise function. If $\varphi : M \to W$ is a representation of M over field $\mathbb{F}$, and M' is the matroid derived from M by adding $y \notin E(M)$ as a coloop, we can construct a representation $\varphi' : M' \to (W \times \mathbb{F})$ via a piecewise function as

$$\begin{array}{c} \begin{matrix} a & b & c & d & e & f \end{matrix} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 2 \\ 0 & 1 & 2 & 1 & 2 & 0 \\ 2 & 0 & 1 & 1 & 0 & 1 \end{pmatrix} \end{array} \xrightarrow{\text{adding coloop } x} \begin{array}{c} \begin{matrix} a & b & c & d & e & f & x \end{matrix} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 2 & 0 \\ 0 & 1 & 2 & 1 & 2 & 0 & 0 \\ 2 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

Figure 2.4: If we have a matrix representation A of M/x, and x is a coloop of M, we can extend A to a representation of M by adding a new row and a new column with all zero entries except the entry corresponding to x.

follows: For $e \in E(M')$, we have

$$\varphi'(e) = \begin{cases} (0,1) & e = y \\ (\varphi(e), 0) & e \neq y \end{cases}$$

Our implementation of this piecewise function uses an `if-then-else` statement, as well as `linear_map.inl` and `linear_map.inr`, which are the embeddings of W and $\mathbb{F}$ into $W \times \mathbb{F}$, respectively.

```
/-- If `M` has a representation over 𝔽 and `f : α` is not an element of `
    M.E`, we can construct a representation of the matroid created by
    adding `f` as a coloop to `M` -/
def add_coloop_rep (φ : rep 𝔽 W M) {f : α} (hf : f ∉ M.E) :
  rep 𝔽 (W × 𝔽) (add_coloop M f) := {
    /- This function maps e to (0, 1) when e = f and (φ e, 0)
    otherwise. -/
    to_fun := λ (e : α), if e ∈ ({f} : set α)
        then linear_map.inr 𝔽 W 𝔽 ((λ e : α, 1) e)
        else linear_map.inl 𝔽 W 𝔽 (φ e),
    valid' := [...],
    support := [...] }
```

This section allowed us to derive the following lemmas:

```
/-- If `M` is an excluded minor for representability, then no element of `
    M.E` can be a loop -/
lemma excluded_minor_nonloop (M : matroid α) [finite_rk M]
(hM : excluded_minor matroid.is_binary M) {f : α} (hf : f ∈ M.E) :
   M.nonloop f := [...]
```

```
/-- If `M` is an excluded minor for representability, then no element of `
   M.E` can be a coloop, i.e. single element cocircuit -/
lemma excluded_minor_noncoloop (M : matroid α) [finite_rk M]
(hM : excluded_minor {N : matroid α | N.is_representable 𝔽} M)
{y : α} (hf : y ∈ M.E) :
  ¬ M.cocircuit {y} := [...]
```

2.6.2 Parallel Extension

In a similar vein to our proof that an excluded minor for representation cannot contain loops, it is mathematically easy to show that it cannot contain a parallel pair, i.e. a 2-element circuit. To see why, suppose that x, y is a parallel pair of M, an excluded minor for representability over $\mathbb{F}$. Then, by minimality of M, there exists some representation φ of $M\backslash y$, which we can extend it to a representation of M, contradicting the non-representability of M. In other words, if $\{x, y\}$ is our 2-element circuit, we have some representation of $M\backslash y$, and can construct a representation of the parallel extension of x in $M\backslash y$ by y, which is equal to M.

The representation of a parallel extension of x by y in matroid M is essentially done by copying the column corresponding to x and assigning it to y (or a scalar multiple of it).

$$\begin{array}{c} \begin{matrix} a & b & c & d & e & x \end{matrix} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 2 \\ 0 & 1 & 2 & 1 & 2 & 0 \\ 2 & 0 & 1 & 1 & 0 & 1 \end{pmatrix} \end{array} \xrightarrow[of\ x\ by\ y]{parallel\ extension} \begin{array}{c} \begin{matrix} a & b & c & d & e & x & y \end{matrix} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 2 & 2 \\ 0 & 1 & 2 & 1 & 2 & 0 & 0 \\ 2 & 0 & 1 & 1 & 0 & 1 & 1 \end{pmatrix} \end{array}$$

Figure 2.5: If we want to extend $x \in E$ by $y \notin E$ in some matroid M represented by matrix A, one possible representation of the extension involves copying the column for x and assigning it to y.

For our definition of the representation of a parallel extension, we were able to use our third definition of representability with `matroid_of_module_fun` to simplify the amount of work we needed to do for the proof. Our representation for the parallel extension of a matroid is the following `if-then-else` function. The following code snippet does not appear in our formalization like this, but we have included it for the discussion about the function definition.

```
def parallel_rep_fun : α → W :=
    (λ (e : α), if e = y then - φ x else φ e)
```

In other words, if we are taking a parallel extension of `x` by adding element `y`, and we have `φ : rep 𝔽 W M`, we simply map `y` to `(- φ x)` so that the image of $\{x, y\}$ is linearly dependent. We decided to multiply `φ x` by the scalar `-1`, or the negative of the multiplicative identity in field $\mathbb{F}$, as this makes our calculations easier later.

Now we prove that the matroid defined by our function is equal to the parallel extension:

```
lemma parallel_extend_rep (φ : rep 𝔽 W M) {x y : α} (hMx : M.nonloop x)
   (hy : y ∉ M.E) [finite_dimensional 𝔽 W] :
  matroid_of_module_fun 𝔽 W (λ (e : α), if e = y then - φ x else φ e)
   (insert y M.E) = parallel_extend M x y := [...]
```

Using our lemma `parallel_extend_rep`, if $\{x, y\}$ is a two-element circuit of M, then given a representation of $M \backslash y$, we can construct a representation of M. Proving this gives us our object of type `rep 𝔽 W M`, and allows us to conclude the following lemmas:

```
/-- If `M` is an excluded minor for representability in `𝔽`, and `x,y` are
   distinct elements of `M`, then `{x, y}` cannot form a circuit in `M`. -/
lemma excluded_minor_nonpara (M : matroid α) [finite_rk M]
  (hM : excluded_minor matroid.is_binary M) {x y : α} (hxy : x ≠ y) :
  ¬ M.circuit {x, y} := [...]
```

2.6.3 Series Extension

Our last significant challenge we encountered was formalizing the proof that, if M is an excluded minor for representation over a field, and x and y are two elements of the ground set $E(M)$, then the set $\{x, y\}$ is coindependent in M. To show that the set $\{x, y\}$ is coindependent in an excluded minor for $\mathbb{F}$-representability, we had to break it into cases: first, showing that if x or y is a coloop, then M is representable, and then showing that if $\{x, y\}$ is a cocircuit, M must be representable. This fact in Oxley's exposition of the excluded minor characterization of binary matroids is mentioned in one line, and it would be trivial if we had representations of dual matroids defined. This is because showing that it holds for a circuit is relatively easy, and via duality, we would be able to conclude it for cocircuits. But, as with matroid contraction, we had to prove this directly.

When we perform a series extension of x by y, in the new matroid, x, y forms a cocircuit. This means that every base of the set of column vectors of A must intersect non-trivially with $\{x, y\}$. Therefore, we can think of the representation of a series extension as the

addition of a new row and column to the matrix such that x, the element being extended, and y the element we are adding, have an additional entry that puts them in a new dimension of the vector space. For example, suppose that matroid M with $x \in E(M)$ and $y \notin E(M)$ is represented by the following matroid:

$$\begin{array}{cccccc} a & b & c & d & e & x \\ \end{array}$$
$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 2 \\ 0 & 1 & 2 & 1 & 2 & 0 \\ 2 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

Then we can modify A by first creating a new row of all zeros and a new column for y of all zeros.

$$\begin{array}{ccccccc} a & b & c & d & e & x & y \\ \end{array}$$
$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 2 & 0 \\ 0 & 1 & 2 & 1 & 2 & 0 & 0 \\ 2 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Now, we perform the series extension by changing the zeroes in the new row and in the columns corresponding to x and y into 1's:

$$\begin{array}{ccccccc} a & b & c & d & e & x & y \\ \end{array}$$
$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 2 & 0 \\ 0 & 1 & 2 & 1 & 2 & 0 & 0 \\ 2 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

This is the opposite of the operation we performed for contraction in section 2.5.2, where we deleted a non-zero row and the corresponding column of the element being contracted.

The way we implemented this in our formalization is by saying that if M is representable over field $\mathbb{F}$ in some module W, we can construct a new module $(W \times \mathbb{F})$ to represent the series extension of M. Then, if $\varphi : M \to W$ is a representation of M over field $\mathbb{F}$, and M' is the series extension of M by $y \notin E(M)$ along x, we can construct a representation $\varphi' : M' \to (W \times \mathbb{F})$ via a piecewise function as follows: For $e \in E(M')$, we have

$$\varphi'(e) = \begin{cases} (\varphi(e), 1) & e = x \\ (0, 1) & e = y \\ (\varphi(e), 0) & e \neq x, y \end{cases}$$

In lean, our implementation of this looks like

```
/-- If `φ` is a representation of `M` over `𝔽`, `x ∈ M.E` such that `x` is
    not a coloop of `M`, and `y ∉ M.E`, then we can construct a
    representation of `series_extend M x y` over `𝔽`. -/
def series_extend_rep (φ : rep 𝔽 W M) {x y : α} (hx : x ∈ M.E)
  (hy : y ∉ M.E) (hMx : ¬ M.coloop x) : rep 𝔽 (W × 𝔽) (series_extend M x
   y) := {
    to_fun := λ (e : α),
    if e = x
    then /- e = x, so we map e to (φ(e), 1) -/
      (linear_map.inl 𝔽 W 𝔽 ∘ φ + linear_map.inr 𝔽 W 𝔽 ∘ (λ e : α, 1)) e
    else
        if e = y
        then /- e = y, so we map e to (0, 1) -/
            linear_map.inr 𝔽 W 𝔽 1
        else /- e is not y or x, so we map e to (φ(e), 0) -/
            (linear_map.inl 𝔽 W 𝔽 ∘ φ) e,
    valid' := [...],
    support := [...] }
```

where `linear_map.inl` and `linear_map.inr` are the left and right injections into the product $W \times \mathbb{F}$, treated as linear maps. The definition that uses this function produces an object of type `rep 𝔽 (W × 𝔽) (series_extend M x y)`, which is what we want.

When we perform a series extension of $x \in E(M)$ by $y \notin E(M)$ when x is not a coloop of M, we are creating a cocircuit $\{x, y\}$ of M', which is the inverse operation of contracting y.

This section, as well as `excluded_minor_noncoloop`, gives us the following lemma:

```
/-- If `M` is an excluded minor for representability over a field `𝔽`, and
    `x, y` are two distinct elements of `M.E`, then `{x, y}` must be
    coindependent in `M`. -/
lemma coindep_excluded_minor (M : matroid α)
(hM : excluded_minor {N : matroid α | N.is_representable 𝔽} M)
(x y : α) (hxy : x ≠ y) (hx : {x, y} ⊆ M.E)
  : M.coindep {x, y} := [...]
```

Chapter 3

Excluded Minor Theorem

3.1 Tutte's Theorem & Proof

Tutte's seminal theorem in the 1950's characterizes matroids representable over $GF(2)$ by non-containment of $U_{2,4}$ as a matroid minor [30]. The following proof of the theorem is loosely paraphrased from Oxley's Matroid Theory [25, p.193-194] The structure of this proof is what our formalization follows, and it is broken down into much greater detail than one might expect for that purpose.

Lemma 3.1.1. *Let M be an excluded minor for binary matroids. Then M has no 1- or 2-element circuits.*

Proof. If M had a one-element circuit $\{y\}$, one could obtain a $GF(2)$-representation of M by appending a zero-column to a representation of $M\backslash y$. If M had a two-element circuit $\{x, y\}$, one could obtain a $GF(2)$-representation of M by appending a copy of the column for f to a representation of $M\backslash\{y\}$. □

Lemma 3.1.2. *Let M be an excluded minor for binary matroids. Then M has no 1- or 2-element cocircuits.*

Proof. If M had a one-element cocircuit $\{y\}$, one could obtain a $GF(2)$-representation of M by appending a zero-row and a zero-column to a representation of $M\backslash y$, and changing the entry in the intersection of the new row and column to 1. If M had a two-element

circuit $\{x, y\}$, one could obtain a $GF(2)$-representation of M by modifying a representation of $M\backslash\{y\}$ as follows: append an all-zero column for y and an all-zero row, and change the entries corresponding to x and y in the new row to 1. □

For more details on the representations in Lemmas 3.1.1 and 3.1.2, refer to Sections 2.6.1, 2.6.2, and 2.6.3.

Lemma 3.1.3. *$U_{2,4}$ is an excluded minor for the class of binary matroids.*

Proof. A matroid is an excluded minor for a class of matroids if it is a minor minimal non-element of the class. We can prove that a matroid $M = (E, \mathcal{I})$ is an excluded minor for representability over $\mathbb{F}$ if we show that it is not representable over $\mathbb{F}$, and for every element $e \in E$, $M\backslash e$ and M/e are representable over $\mathbb{F}$. Let $e \in E(U_{2,4})$. Observe that $U_{2,4}\backslash e = U_{2,3}$ and $U_{2,4}/e = U_{1,3}$. For proofs that $U_{2,4}$ is not binary, and $U_{2,3}$ and $U_{1,3}$ are binary, see Section 2.4.1. □

Lemma 3.1.4. *If M_I and M_C are two matroids with the same rank and ground set, $Z \subseteq J \subseteq E(M_I)$, and $x, y \in Z$ are distinct, if*

(i) *$\{x, y\}$ is coindependent in M_I or M_C*

(ii) *$M_I\backslash x = M_C\backslash x$ and $M_I\backslash y = M_C\backslash y$*

(iii) *Z is independent in M_I and dependent in M_C*

(iv) *J is independent in M_I*

then $J = \{x, y\}$.

Proof. First, we know that $x \in Z$ because if it wasn't, then Z is independent in $M_I\backslash x$. Since $M_I\backslash x = M_C\backslash x$ by *(ii)*, we would find that Z is independent in M_C, which is a contradiction of *(iii)*. By symmetry, $y \in Z$, so we conclude that $\{x, y\} \subseteq J$.

Suppose for contradiction that $J \neq \{x, y\}$. Then $J\backslash\{x, y\}$ is nonempty. Furthermore, since J is independent in M_I by *(iv)*, we know that $J\backslash\{x, y\}$ is independent in $M_I\backslash\{x, y\}$, and because $M_I\backslash\{x, y\} = M_C\backslash\{x, y\}$, it is also independent in $M_C\backslash\{x, y\}$.

Let $N_I := M_I/(J\backslash\{x, y\})$ and $N_C := M_C/(J\backslash\{x, y\})$. Because $M_I \neq M_C$, we know that $N_I \neq N_C$, but we will have $r(N_I) = r(N_C)$. Then if B_N is a base of $N_I \backslash\{x, y\}$, either B_N is a base of both N_I and N_C, or neither. Consider the case where B_N is not a base for

either N_I or N_C. Then that would imply that $r(N_I \backslash \{x, y\}) < r(N_I)$, so $\{x, y\}$ contains a cocircuit of N_I. By symmetry, $\{x, y\}$ also contains a cocircuit of N_C. Then $\{x, y\}$ contains a cocircuit of both M_I and M_C, which contradicts condition *(i)*.

Now consider the case where B_N is a base for both N_I and N_C. Since $N_I \backslash x = N_C \backslash x$ and $N_I \backslash y = N_C \backslash y$, the fundamental circuits are equal, $C_{N_I}(e, B_N) = C_{N_C}(e, B_N)$, for all e in $E(N_I) - B$. If M, M' are binary matroids such that $E(M) = E(M')$, $B \subseteq E(M)$ is a base of both M and M', and for all $e \in E(M)$, $C_M(e, B) = C_{M'}(e, B)$, then $M = M'$. Then $N_I = N_C$, a contradiction. Therefore, $J \backslash \{x, y\}$ cannot be nonempty, and we have $J = \{x, y\}$. □

Lemma 3.1.5. *If M is an excluded minor for binary representation, then $M = U_{2,4}$.*

Proof. Let x, y be distinct elements of M. Then by Lemma 3.1.1, $\{x, y\}$ cannot contain a one- or two-element cocircuit of M. Then if x, y are distinct elements of M, we have $r(M \backslash \{x, y\}) = r(M)$, so there exists a base B of $M \backslash \{x, y\}$ that is also a base of M. Suppose that the matrix $[I_B|D]$ represents $M \backslash \{x, y\}$ over $GF(2)$, where I_B is the identity matrix indexed by B. Since M is minor minimal with respect to representation over $GF(2)$, $M \backslash x$ and $M \backslash y$ are binary.

Let v_x be the sum of columns of I_B that correspond to elements of B that are contained in the fundamental circuit $C(x, B)$. Then $[I_B|D|v_x]$ is a representation of $M \backslash y$ over $GF(2)$ (this fact relies on the uniqueness of matroid representations over $GF(2)$ and is not true for general fields). Similarly, if v_y is the sum of columns of I_B that correspond to elements of B contained in $C(y, B)$, the matrix $[I_B|D|v_y]$ is a representation of $M \backslash x$ over $GF(2)$.

Now, let M' be the matroid represented by $[I_B|D|v_x|v_y]$ over $GF(2)$. Then $M \backslash x = M' \backslash y$ and $M \backslash y = M' \backslash y$. Furthermore, $r(M) = r(M')$, so B is also a base for M'. Since $M \neq M'$, there exists some set $Z \subseteq E(M)$ such that either Z is independent in M and dependent in M', or vice versa. Let M_I be the matroid out of M, M' for which Z is independent, and M_C be the matroid out of M, M' for which Z is dependent.

We know that $\{x, y\}$ is coindependent in M, so that means $\{x, y\}$ is coindependent in M_I or M_C. Furthermore, we have $M_I \backslash x = M_C \backslash x$ and $M_I \backslash y = M_C \backslash y$. From our previous paragraph, we have that Z is independent in M_I and dependent in M_C, and let J be an independent set of M_I such that $Z \subseteq J$. Then by Lemma 3.1.4, $J = \{x, y\}$, and therefore $r(M_I) = 2$.

We established that $r(M) = r(M')$, and $r(M_I) = 2$, so we may conclude that $r(M) = r(M') = 2$. By Lemma 3.1.1, we know that M cannot contain 1- or 2-element circuits, so it must be simple. Because M is a simple matroid of rank 2, it must be a uniform matroid

of rank 2. By Lemma 3.1.3, $U_{2,4}$ is an excluded minor for binary matroids, so it is the minor-minimal uniform matroid of rank 2 that is not binary. Thus, $M = U_{2,4}$. □

We recall theorem 1.4.1, and now we are ready to provide a proof:

Theorem 1.4.1 (Tutte's 1958 excluded minor characterization of binary matroids)**.** *A matroid is binary if and only if it has no $U_{2,4}$-minor.*

Proof. ($\rightarrow$) Let M be a binary matroid. Assume for contradiction that $U_{2,4}$ is a minor of M. Then since a minor of a binary matroid must be binary, $U_{2,4}$ must be binary, a contradiction.

($\leftarrow$) Let M be a matroid with no $U_{2,4}$-minor, and suppose for contradiction that M is not binary. Then M must have some excluded minor M' for binary representation. By Lemma 3.1.5, $M' = U_{2,4}$, a contradiction. Therefore, M is binary. □

3.2 Rank of Excluded Minor

Most of the proofs in this section are rather long and technical, so we omit them here but provide them in Appendix A (as well as in the Github repository for this project).

If φ is a representation of `M` over `zmod 2`, and `I` is an independent set of `M`, there is a unique way of writing any element `e` in the span of `I` as the sum of the vectors corresponding to the intersection of `I` with the fundamental circuit of `e` and `I`. We use this sum to define a matroid corresponding to M' in our formalized proof of Lemma 3.1.5.

```
/-- If `φ` is a representation of `M` in `zmod 2`, `I` is an independent set
    of `M`, and `e` is a member of the closure of `I`, then `φ e` can be
    uniquely written as the sum of `φ i` for `i` contained in the
    intersection of the fundamental circuit of `I` and `e`, and `I` -/
lemma mem_sum_basis_zmod2 [fintype α] [module (zmod 2) W] (φ : rep (zmod
    2) W M) {I : set α} (hI : M.indep I) (e : α) (he : e ∈ M.cl I) :
  φ e = Σ i in (M.fund_circuit e I ∩ I).to_finset, φ i := [...]

/-- If `M` is a matroid, if
    - `B : set α` such that `B` is a base for `M \ {x, y}`, `M \ x`, and `M`
    - `φ` is a representation of `M \ {x, y}` over `zmod 2`
    - `φx` is a representation of `M \ x` over `zmod 2`
    then `(M \ x) = (matroid_of_module_fun (zmod 2) (B →₀ zmod 2)`
```

```
    (λ e : α, Σ i in (M.fund_circuit e B ∩ B), (standard_rep φ hBxy) i)
   M.E) \ x`. -/
lemma delete_elem_eq_of_binary {B : set α} {x y : α}
(hBxy : (M \ {x, y}).base B)
(hBx : (M \ x).base B) (hB : M.base B)
(φ : rep (zmod 2) W (M \ {x, y}))
(φx : rep (zmod 2) Wx (M \ x)) :
(M \ x) = (matroid_of_module_fun (zmod 2) (B →₀ zmod 2)
(λ e : α, Σ i in (M.fund_circuit e B ∩ B),
(standard_rep φ hBxy) i) M.E) \ x := [...]
```

The following `lemma` corresponds to Lemma 3.1.4.

```
/-- If `MI` and `MC` are two matroids with the same rank and ground set,
`Z ⊆ J ⊆ MI.E`, and `x, y ∈ Z` are distinct, if
    - `{x, y}` is coindependent in `MI` or `MC`
    - `MI \ x = MC \ x` and `MI \ y = MC \ y`
    - `Z` is independent in `MI` and dependent in `MC`
    - `J` is independent in `MI`
    then `J = {x, y}`. -/
lemma indep_eq_doubleton_of_subset [fintype α] (MI MC : matroid_in α)
   [finite_rk MI] [finite_rk MC]
  (hrk : MI.rk = MC.rk) (hIC : MI.E = MC.E) (x y : α) (hxy : x ≠ y)
  (hiIC : MI.coindep {x,y} ∨ MC.coindep {x,y}) (hMx : MI \ x = MC \ x)
   (hMy : MI \ y = MC \ y)
  {Z J : set α} (hxZ : x ∈ Z) (hyZ : y ∈ Z) (hMIZ : MI.indep Z) (hMCZ : ¬
    MC.indep Z)
  (hZJ : Z ⊆ J) (hMIJ : MI.indep J) [module (zmod 2) W] [module (zmod 2)
   W']
  (φI : rep (zmod 2) W (MI / (J \ {x, y})))
  (φC : rep (zmod 2) W' (MC / (J \ {x, y}))) : J = {x, y} := [...]
```

In the last part of our proof of Lemma 3.1.4, we show that $N_I = N_C$ due to the fact that, for base B of N_I, N_C, for all $e \in E(N_I)$, we have $C_{N_I}(e) = C_{N_C}(e)$. The following lemma is our formalization of this, which uses `mem_sum_basis_zmod2` in its proof:

```
/-- If `M, M'` are matroids with equal ground sets with representations `φ
   M, φM'` in `zmod 2`, `B : set α` is a `base` of `M` and `M'`, and for all `
   e ∈ M.E`, `M.fund_circuit e B = M'.fund_circuit e B`, then `M = M'`  -/
lemma eq_of_forall_fund_circuit_eq [fintype α] {M M' : matroid_in α}
   [module (zmod 2) W]
[module (zmod 2) W'] (φM : rep (zmod 2) W M) (φM' : rep (zmod 2) W' M')
(hE : M.E = M'.E) (hB : M.base B) (hB' : M'.base B)
```

```
(he : ∀ e ∈ M.E, M.fund_circuit e B = M'.fund_circuit e B) :
  M = M' := [...]
```

The rest of our formalization has to do with the final conclusions in the proof of Lemma 3.1.5. Following Oxley's proof, we show that if M is an excluded minor for $\mathbb{F}_2$-representability, it must have rank 2, and since excluded minors must be simple, we know that M must be isomorphic to $U_{2,k}$ for some k. The bulk of the proof has to do with proving that the matroid has rank 2. Because we invoke smaller lemmas in the proof of 3.1.5, we delayed stating it until this point.

Our formal proof of Lemma 3.1.5 uses lemmas `excluded_minor_nonpara` from Section 2.6.2, which corresponds to Lemma 3.1.1, and `coindep_excluded_minor` from Section 2.6.3, which corresponds to Lemma 3.1.2. Additionally, we use our `U24_excluded_minor` lemma from Section 2.4.1, as well as the lemmas outlined above in this section.

```
/-- If `M is an excluded minor for binary representation, then `M has
    rank 2 -/
lemma excluded_minor_binary_rk2 (M : matroid_in α) [finite_rk M]
  (hM : excluded_minor (set_of matroid_in.is_binary) M) : M.rk = 2 := [...]

/-- If `M is an excluded minor for binary representation, then `unif 2 4
    ≤i M -/
lemma excluded_minor_binary (M : matroid_in α) [finite_rk M]
  (hM : excluded_minor (set_of matroid_in.is_binary) M) : unif 2 4 ≤i M :=
    [...]

/-- If `M is an excluded minor for binary representation, then `M is
    isomorphic to `unif 2 4`. This corresponds to Lemma 3.1.5. -/
lemma excluded_minor_binary_iso_unif24 (M : matroid_in α) [finite_rk M]
  (hM : excluded_minor (set_of matroid_in.is_binary) M) : nonempty (M ≃i
    (unif 2 4)) := [...]
```

We restate theorem 1.4.1 before our formalized theorem statement and proof.

Theorem 1.4.1 (Tutte's 1958 excluded minor characterization of binary matroids)**.** *A matroid is binary if and only if it has no $U_{2,4}$-minor.*

This fundamental theorem was proven by Tutte in 1958 and has been formalized. This lays a foundation for future matroid representation work, such as the more difficult excluded minor theorems for fields like $GF(3)$ and $GF(4)$, as well as the excluded minor characterization of graphic matroids.

Finally, the formalized theorem statement and proof:

```
/-- Tutte's 1958 theorem, which states that a matroid is binary if and
    only if it contains no `unif 2 4` minor. -/
theorem binary_iff_no_u24_minor (M : matroid_in α) [finite_rk M] :
  matroid_in.is_binary M ↔ ¬ unif 2 4 ≤i M :=
begin
/- Split the iff statement into two directions, and simplify the second
    direction with `mem_iff_no_excluded_minor_minor`, which says `M` is
    binary iff no excluded minor for binary matroids is a minor of `M` -/
  refine ⟨λ hfM, _, λ h3, (@mem_iff_no_excluded_minor_minor _ M _
    (matroid_in.is_binary)
     (@minor_closed_rep _ (zmod 2) _)).2 (λ M' hM', _)⟩,
    /- The forward direction, with hypothesis `hfM :
    matroid_in.is_binary M` -/
  { /- Assume for contradiction that `unif 2 4 ≤i M` -/
    by_contra,
    /- Obtain the minor `M'` of `M` such that `unif 2 4 ≃i M'`-/
    obtain ⟨M', ⟨hM', ⟨ψ⟩⟩⟩ := h,
    /- Derive a contradiction by concluding that `unif 2 4` is binary -/
    apply ((excluded_minor_iff (set_of matroid_in.is_binary)
    (@minor_closed_rep _ (zmod 2) _)).1
       ((excluded_minor_binary_iff_iso_unif24 M').2 ⟨ψ.symm⟩)).1
       (is_rep_of_minor_of_is_rep _ hM' hfM) },
    /- The backwards direction, which we have already partially
    simplified -/
  { /- Assume for contradiction that some excluded minor `M'` for binary
    matroids is a minor of `M`, so that `M` is not binary -/
    by_contra,
    /- Use `excluded_minor_binary_iso_unif24` to obtain an isomorphism `ψ`
    between `M'` and `unif 2 4` -/
    obtain ⟨ψ⟩ := excluded_minor_binary_iso_unif24 M' hM',
    /- Conclude a contradiction, as `unif 2 4` is isomorphic to a minor
    of `M`, but we assumed it is not an `iso_minor` of `M` -/
    refine h3 ⟨M', ⟨h, ⟨ψ.symm⟩⟩⟩ },
end
```

References

[1] *A Lost Mathematician, Takeo Nakasawa*. Birkhäuser Basel, 2009.

[2] Jeremy Avigad, Leonardo de Moura, and Soonho Kong. Theorem Proving in Lean, 2017.

[3] Jeremy Avigad, Leonardo de Moura, Soonho Kong, and Sebastian Ullrich. Theorem Proving in Lean 4.

[4] Henning Bruhn, Reinhard Diestel, Matthias Kriesell, Rudi Pendavingh, and Paul Wollan. Axioms for infinite matroids. *Advances in Mathematics*, 239:18–46, June 2013.

[5] Kevin Buzzard, Johan Commelin, and Patrick Massot. Formalising perfectoid spaces. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, page 299–312, New York, NY, USA, 2020. Association for Computing Machinery.

[6] Bryan Gin-ge Chen. lean-matroids. https://github.com/bryangingechen/lean-matroids, 2019.

[7] Mathlib Community. Completion of the liquid tensor experiment, July 2022.

[8] William J. Cook, William H. Cunningham, William R. Pulleyblank, and Alexander Schrijver. *Combinatorial Optimization*. Wiley, November 1997.

[9] N. G. de Bruijn. *AUTOMATH, a Language for Mathematics*, pages 159–200. Springer Berlin Heidelberg, 1983.

[10] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean Theorem Prover (system description). In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25 - 25th International*

Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings, volume 9195 of *Lecture Notes in Computer Science*, pages 378–388. Springer, Jun 2015.

[11] Yaël Dillies and Bhavik Mehta. Formalising szemerédi's regularity lemma in lean. In June Andronick and Leonardo de Moura, editors, *International Conference on Interactive Theorem Proving (ITP)*, volume 237 of *Leibniz International Proceedings in Informatics (LIPIcs)*. Schloss-Dagstuhl - Leibniz Zentrum für Informatik, August 2022.

[12] Christian Doczkal and Damien Pous. Graph theory in Coq: minors, treewidth, and isomorphisms. *Journal of Automated Reasoning*, 64(5):795–825, jan 2020.

[13] Chelsea Edmonds, Angeliki Koutsoukou-Argyraki, and Lawrence C. Paulson. Formalising szemerédi's regularity lemma and roth's theorem on arithmetic progressions in isabelle/hol. *Journal of Automated Reasoning*, 67, March 2023.

[14] Georges Gonthier. Formal proof—the four-color theorem. *Notices Amer. Math. Soc.*, 55(11):1382–1393, 2008.

[15] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley, Reading, Massachusetts, 1994.

[16] Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF*. Springer Berlin Heidelberg, 1979.

[17] W. T. Gowers, Ben Green, Freddie Manners, and Terence Tao. On a conjecture of marton. November 2023.

[18] Thomas Hales. Big conjectures (talk), July 2017.

[19] Eric Katz. Matroid theory for algebraic geometers, 2014.

[20] Sebastian Ullrich Leonardo de Moura. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction - CADE-28 - 28th International Conference on Automated Deduction*, CADE-28, 2021.

[21] Nicolas Magaud. Integrating an automated prover for projective geometry as a new tactic in the coq proof assistant. *EPTCS 336, 2021, pp. 40-47*, 336:40–47, July 2021.

[22] mathlib Community. algebra.module.basic - mathlib3 docs.

[23] mathlib Community. The Lean mathematical library.

[24] Bhavik Mehta. *Formalising the Kruskal-Katona Theorem in Lean*, pages 75–91. Springer International Publishing, 2022.

[25] James Oxley. *Matroid Theory*. Oxford University Texts, 2nd edition, 2011.

[26] Jonathan Prieto-Cubides. On homotopy of walks and spherical maps in homotopy type theory. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP '22. ACM, January 2022.

[27] Neil Robertson, Daniel Sanders, Paul Seymour, and Robin Thomas. The four-colour theorem. *Journal of Combinatorial Theory, Series B*, 70(1):2–44, May 1997.

[28] Terence Tao. Formalizing the proof of PFR in Lean4 using Blueprint: a short tour. Blog post, November 2023.

[29] mathlib community Terence Tao. teorth. https://github.com/teorth/pfr, 2023.

[30] W. T. Tutte. A homotopy theorem for matroids. i, ii. *Transactions of the American Mathematical Society*, 88(1):144–174, May 1958.

[31] W.T. Tutte. Lectures on matroids. *Journal of Research of the National Bureau of Standards Section B Mathematics and Mathematical Physics*, 69B(1 and 2):1, January 1965.

[32] Floris van Doorn, Patrick Massot, and Oliver Nash. Formalising the h-principle and sphere eversion. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP '23. ACM, January 2023.

[33] Hassler Whitney. On the abstract properties of linear dependence. *American Journal of Mathematics*, 57(3):509, July 1935.

[34] Freek Wiedijk. Formalizing 100 theorems (webpage).

[35] Mitsuharu Yamamoto, Shin-ya Nishizaki, Masami Hagiya, and Yozo Toda. *Formalization of planar graphs*, pages 369–384. Springer Berlin Heidelberg, 1995.

APPENDICES

Appendix A

Minimum Working Source Code

The full project, and instructions for how to install `lean` and run it on your computer, can be found on Github[1].

[1] https://github.com/agusakov/excluded_minor_binary

A.1 Preliminary Lemmas and Definitions

```
/-
Code copied from https://github.com/apnelson1/lean-matroids
-/
import data.set.image
import data.set.ncard
import data.set.function
import data.set.basic
import data.nat.lattice
import order.minimal
import data.finset.locally_finite
import data.nat.interval


section minimal

variables {α : Type*} {r : α → α → Prop} {s : set α} {x y : α} {P : α
   → Prop}

lemma mem_maximals_iff' [is_antisymm α r] :
  x ∈ maximals r s ↔ x ∈ s ∧ ∀ ⦃y⦄, y ∈ s → r x y → x = y :=
begin
  simp only [maximals, set.mem_sep_iff, and.congr_right_iff],
  refine λ hx, ⟨λ h y hys hxy, antisymm hxy (h hys hxy), λ h y hys hxy,
   _⟩,
  convert hxy; rw h hys hxy,
end

lemma mem_minimals_iff' [is_antisymm α r] :
  x ∈ minimals r s ↔ x ∈ s ∧ ∀ ⦃y⦄, y ∈ s → r y x → x = y :=
by { convert mem_maximals_iff', apply is_antisymm.swap }

lemma mem_maximals_prop_iff [is_antisymm α r] :
  x ∈ maximals r P ↔ P x ∧ ∀ ⦃y⦄, P y → r x y → x = y :=
mem_maximals_iff'

lemma mem_maximals_set_of_iff [is_antisymm α r] :
  x ∈ maximals r (set_of P) ↔ P x ∧ ∀ ⦃y⦄, P y → r x y → x = y :=
mem_maximals_iff'

lemma mem_minimals_prop_iff [is_antisymm α r] :
```

```
  x ∈ minimals r P ↔ P x ∧ ∀ {|y|}, P y → r y x → x = y :=
mem_minimals_iff'

lemma mem_minimals_set_of_iff [is_antisymm α r] :
  x ∈ minimals r (set_of P) ↔ P x ∧ ∀ {|y|}, P y → r y x → x = y :=
mem_minimals_iff'

end minimal

open set

section finite -- taken from mathlib.data.set.finite.lean

namespace set

variables {α β : Type*}

lemma finite.exists_maximal [finite α] [partial_order α] (P : α → Prop)
   (h : ∃ x, P x) :
  ∃ m, P m ∧ ∀ x, P x → m ≤ x → m = x :=
begin
  obtain ⟨m, hm,hm'⟩ := finite.exists_maximal_wrt (@id α) (set_of P)
   (to_finite _) h,
  exact ⟨m, hm, hm'⟩,
end

end set

end finite

section ncard -- taken from mathlib.data.set.ncard.lean

variables {α : Type*} {s t r : set α} {x y z : α}

namespace set

lemma enat.coe_inj {m n : ℕ} : (m : ℕ∞) = n ↔ m = n :=
nat.cast_inj

lemma enat.coe_le_coe_iff {m n : ℕ} : (m : ℕ∞) ≤ n ↔ m ≤ n :=
nat.cast_le
```

```
/-- The cardinality of a set as a member of `enat`. -/
noncomputable def encard (s : set α) : ℕ∞ := part_enat.with_top_equiv
    (part_enat.card s)

lemma finite.encard_eq (hs : s.finite) : s.encard = (s.ncard : ℕ∞) :=
begin
  obtain ⟨s, rfl⟩ := hs.exists_finset_coe,
  simp_rw [encard, part_enat.card_eq_coe_fintype_card,ncard_coe_finset,
    part_enat.with_top_equiv_coe, nat.cast_inj, finset.coe_sort_coe,
   fintype.card_coe],
end

lemma infinite.encard_eq (hs : s.infinite) : s.encard = ⊤ :=
begin
  haveI := hs.to_subtype,
  rw [encard, part_enat.card_eq_top_of_infinite,
   part_enat.with_top_equiv_top],
end

@[simp] lemma encard_to_nat_eq (s : set α) : s.encard.to_nat = s.ncard :=
begin
  obtain (h | h) := s.finite_or_infinite,
  simp [h.encard_eq],
  simp [h.ncard, h.encard_eq],
end

lemma encard_insert_of_not_mem (h : x ∉ s) : (insert x s).encard =
   s.encard + 1 :=
begin
  obtain (hs | hs) := s.finite_or_infinite,
  { rw [hs.encard_eq, (hs.insert x).encard_eq, ncard_insert_of_not_mem h
   hs], simp },
  rw [hs.encard_eq, (hs.mono (subset_insert _ _)).encard_eq,
   with_top.top_add],
end

@[simp] lemma encard_eq_top_iff_infinite : s.encard = ⊤ ↔ s.infinite :=
begin
  refine ⟨λ h hfin, _, infinite.encard_eq⟩,
  rw hfin.encard_eq at h,
  simpa only [with_top.nat_ne_top] using h,
end
```

```
@[simp] lemma encard_lt_top_iff_finite : s.encard < ⊤ ↔ s.finite :=
by rw [lt_top_iff_ne_top, ←not_infinite, ←encard_eq_top_iff_infinite]

lemma finite_of_encard_le_coe {k : ℕ} (h : s.encard ≤ k) : s.finite :=
encard_lt_top_iff_finite.mp (h.trans_lt (with_top.coe_lt_top k))

lemma encard_le_coe_iff {k : ℕ} : s.encard ≤ k ↔ s.finite ∧ s.ncard ≤
   k :=
begin
  refine ⟨λ h, _, λ h, _⟩,
  { have hs := finite_of_encard_le_coe h,
    rw [hs.encard_eq, nat.cast_le] at h,
    exact ⟨hs, h⟩ },
  rw [h.1.encard_eq, nat.cast_le],
  exact h.2,
end

lemma encard_union_eq (h : disjoint s t) : (s ∪ t).encard = s.encard +
   t.encard :=
begin
  obtain (hf | hi) := (s ∪ t).finite_or_infinite,
  { obtain ⟨hs, ht⟩ := finite_union.mp hf,
    rw [hf.encard_eq, hs.encard_eq, ht.encard_eq, ←nat.cast_add,
   nat.cast_inj,
      ncard_union_eq h hs ht] },
  rw [hi.encard_eq],
  obtain (h | h) := infinite_union.mp hi; simp [h.encard_eq],
end

lemma encard_diff_add_encard_inter (s t : set α) :
  (s \ t).encard + (s ∩ t).encard = s.encard :=
by rw [←encard_union_eq (disjoint_of_subset_right (inter_subset_right _
   _) disjoint_sdiff_left),
    diff_union_inter]

lemma encard_eq_coe_iff {k : ℕ} : s.encard = k ↔ s.finite ∧ s.ncard = k
   :=
begin
  refine ⟨λ h, _, λ h, _⟩,
  { have hs := finite_of_encard_le_coe h.le,
    rw [hs.encard_eq, nat.cast_inj] at h,
```

```
    exact ⟨hs,h⟩},
  rw [h.1.encard_eq, h.2],
end

lemma encard_subset_le (hst : s ⊆ t) : s.encard ≤ t.encard :=
begin
  obtain (ht | ht) := t.finite_or_infinite,
  { rw [ht.encard_eq, (ht.subset hst).encard_eq, nat.cast_le],
    exact ncard_le_of_subset hst ht },
  exact le_top.trans_eq ht.encard_eq.symm,
end

lemma encard_mono : monotone (encard : set α → ℕ∞) :=
λ _ _, encard_subset_le

lemma encard_image_of_injective {α β : Type*} {f : α → β} (s : set α)
    (hf : f.injective) :
  (f '' s).encard = s.encard  :=
begin
  obtain (hs | hs) := s.finite_or_infinite,
  { rw [hs.encard_eq, (hs.image f).encard_eq, ncard_image_of_injective s
    hf] },
  rw [hs.encard_eq, infinite.encard_eq],
  rwa infinite_image_iff (hf.inj_on _),
end

lemma encard_preimage_of_injective_subset_range {α β : Type*} {f : α → β
    } {s : set β}
  (hf : f.injective) (hs : s ⊆ range f) : (f ⁻¹' s).encard = s.encard :=
begin
  obtain ⟨t, rfl⟩ := subset_range_iff_exists_image_eq.mp hs,
  rw [← encard_image_of_injective _ hf, preimage_image_eq _ hf],
end

end set

end ncard

section image -- taken from mathlib.data.set.image.lean

variables {α β : Type*} {f : α → β}
```

```
@[simp] lemma subtype.preimage_image_coe (s : set α) (x : set s) :
  (coe ⁻¹' (coe '' x : set α) : set s) = x := preimage_image_eq x
   subtype.coe_injective

@[simp] lemma subtype.image_coe_eq_image_coe_iff (s : set α) (x y : set
   s) :
  (coe '' x : set α) = coe '' y ↔ x = y :=
image_eq_image subtype.coe_injective

end image

section basic -- taken from mathlib.data.set.basic.lean

variables {α : Type*} {s t r : set α} {a : α}

namespace set

lemma singleton_inter_eq_of_mem {x : α} (hx : x ∈ s) :
  {x} ∩ s = {x} :=
(inter_subset_left _ _).antisymm (subset_inter subset_rfl
   (singleton_subset_iff.mpr hx))

@[simp] lemma not_mem_diff_singleton {α : Type*} (x : α) (s : set α) :
  x ∉ s \ {x} :=
not_mem_diff_of_mem $ mem_singleton _

lemma ssubset_iff_subset_diff_singleton {α : Type*} {s t : set α} :
  s ⊂ t ↔ ∃ x ∈ t, s ⊆ t \ {x} :=
begin
  refine ⟨λ h, _,_⟩,
  { obtain ⟨x,hxt,hxs⟩ := exists_of_ssubset h,
    refine ⟨x, hxt, subset_diff_singleton h.subset hxs⟩ },
   rintro ⟨x, hxt, hs⟩,
  exact hs.trans_ssubset (diff_singleton_ssubset.mpr hxt),
end

lemma inter_subset_union (s t : set α) : s ∩ t ⊆ s ∪ t :=
(inter_subset_left _ _).trans (subset_union_left _ _)

/-- `r` is an explicit parameter here for ease of rewriting. -/
lemma diff_subset_diff_iff (r : set α) (hs : s ⊆ r) (ht : t ⊆ r) : (r \
   s) ⊆ (r \ t) ↔ t ⊆ s :=
```

```
begin
  rw [subset_diff, and_iff_right (diff_subset _ _), ←
   subset_compl_iff_disjoint_left, diff_eq,
    compl_inter, compl_compl],
  exact ⟨λ h x hxt, (h hxt).elim (λ h', (h' (ht hxt)).elim) id,
    λ h, h.trans (subset_union_right _ _)⟩,
end

lemma diff_eq_diff_iff_inter_eq_inter : r \ s = r \ t ↔ r ∩ s = r ∩ t :=
by simp only [set.ext_iff, mem_diff, and.congr_right_iff, mem_inter_iff,
   not_iff_not]

lemma diff_eq_diff_iff_eq (hsr : s ⊆ r) (htr : t ⊆ r) : r \ s = r \ t ↔
   s = t :=
by rw [diff_eq_diff_iff_inter_eq_inter, inter_eq_self_of_subset_right hsr,
  inter_eq_self_of_subset_right htr]

@[simp] lemma diff_inter_diff_right (s t r : set α) : (s \ r) ∩ (t \ r) =
   (s ∩ t) \ r :=
by { ext, simp only [mem_inter_iff, mem_diff], tauto }

lemma inter_diff_right_comm (s t r : set α) : s ∩ t \ r = (s \ r) ∩ t :=
by rw [diff_eq, inter_right_comm, ←diff_eq]

end set

end basic
```

A.2 Matroid Definitions

```
/-
Code copied from https://github.com/apnelson1/lean-matroids
-/
import .prelim_mwe
import tactic

noncomputable theory
open_locale classical
open_locale big_operators
```

```
open set

section prelim -- taken from basic.lean

variables {α : Type*} {I D J B B' B₁ B₂ X Y : set α} {e f : α}

/-- A predicate `P` on sets satisfies the exchange property if, for all `
   X` and `Y` satisfying `P`
  and all `a ∈ X \ Y`, there exists `b ∈ Y \ X` so that swapping `a` for `b`
   in `X` maintains `P`.-/
def exchange_property (P : set α → Prop) : Prop :=
  ∀ X Y, P X → P Y → ∀ a ∈ X \ Y, ∃ b ∈ Y \ X, P (insert b (X \ {a}))

/-- A predicate `P` on sets satisfies the maximal subset property if, for
   all `X` containing some
  `I` satisfying `P`, there is a maximal subset of `X` satisfying `P`. -/
def exists_maximal_subset_property (P : set α → Prop) (X : set α) : Prop
   :=
  ∀ I, P I → I ⊆ X → (maximals (⊆) {Y | P Y ∧ I ⊆ Y ∧ Y ⊆ X}).nonempty

lemma exists_maximal_subset_property_of_bounded {P : set α → Prop}
(h : ∃ n, ∀ I, P I → (I.finite ∧ I.ncard ≤ n)) (X : set α):
  exists_maximal_subset_property P X :=
begin
  obtain ⟨n,h⟩ := h,
  intros I₀ hI₀ hI₀X,
  set S := {I | P I ∧ I₀ ⊆ I ∧ I ⊆ X} with hS,
  haveI : nonempty S, from ⟨⟨I₀, hI₀, subset.rfl, hI₀X⟩⟩,
  set f : {I | P I ∧ I₀ ⊆ I ∧ I ⊆ X} → fin (n+1) :=
    λ I, ⟨ncard (I : set α), nat.lt_succ_of_le (h I I.2.1).2⟩ with hf,

  obtain ⟨m, ⟨⟨J,hJ⟩,rfl⟩, hJmax⟩ :=  set.finite.exists_maximal (range f)
    (range_nonempty _),
  refine ⟨J, hJ, λ K hK hJK, (eq_of_subset_of_ncard_le hJK _ (h _
    hK.1).1).symm.subset⟩,
  exact (hJmax _ ⟨⟨K,hK⟩, rfl⟩ (ncard_le_of_subset hJK (h _
    hK.1).1)).symm.le,
end

private lemma antichain_of_exch {base : set α → Prop} (exch :
   exchange_property base)
(hB : base B) (hB' : base B') (h : B ⊆ B') : B = B' :=
```

```
begin
  refine h.antisymm (diff_eq_empty.mp (eq_empty_iff_forall_not_mem.mpr (λ
   x hx, _))),
  obtain ⟨e,he,-⟩ :=  exch _ _ hB' hB _ hx,
  exact he.2 (h he.1),
end

private lemma diff_aux_of_exch {base : set α → Prop} (exch :
   exchange_property base)
(hB₁ : base B₁) (hB₂ : base B₂) (hfin : (B₁ \ B₂).finite) :
(B₂ \ B₁).finite ∧ (B₂ \ B₁).ncard = (B₁ \ B₂).ncard :=
begin
  suffices : ∀ {k B B'}, base B → base B' → (B \ B').finite → ncard (B \
   B') = k →
    (B' \ B).finite ∧ (B' \ B).ncard = k, from this hB₁ hB₂ hfin rfl,
  intro k, induction k with k IH,
  { intros B B' hB hB' hfin h0,
    rw [ncard_eq_zero hfin, diff_eq_empty] at h0,
    rw [antichain_of_exch exch hB hB' h0, diff_self],
    simp },
  intros B B' hB hB' hfin hcard,
  obtain ⟨e,he⟩ := nonempty_of_ncard_ne_zero (by { rw hcard, simp } : (B \
   B').ncard ≠ 0),
  obtain ⟨f,hf,hB0⟩ := exch _ _ hB hB' _ he,
  have hef : f ≠ e, by { rintro rfl, exact hf.2 he.1 },

  obtain ⟨hfin',hcard'⟩ := IH hB0 hB' _ _,
  { rw [insert_diff_singleton_comm hef, diff_diff_right,
      inter_singleton_eq_empty.mpr he.2, union_empty, ←union_singleton,
   ←diff_diff] at hfin' hcard',
    have hfin'' := hfin'.insert f,
    rw [insert_diff_singleton, insert_eq_of_mem hf] at hfin'',
    apply_fun (λ x, x+1) at hcard',
    rw [ncard_diff_singleton_add_one hf hfin'', ←nat.succ_eq_add_one] at
   hcard',
    exact ⟨hfin'', hcard'⟩ },
  { rw [insert_diff_of_mem _ hf.1, diff_diff_comm],
    exact hfin.diff _ },
  rw [insert_diff_of_mem _ hf.1, diff_diff_comm,
   ncard_diff_singleton_of_mem he hfin, hcard,
    nat.succ_sub_one],
end
```

```
private lemma finite_of_finite_of_exch {base : set α → Prop} (exch :
   exchange_property base)
(hB : base B) (hB' : base B') (h : B.finite) :
  B'.finite :=
begin
  rw [←inter_union_diff B' B],
  exact finite.union (h.subset (inter_subset_right _ _))
    (diff_aux_of_exch exch hB hB' (h.diff _)).1,
end

/- an `encard` version -/
private lemma encard_diff_eq_of_exch {base : set α → Prop} (exch :
   exchange_property base)
(hB₁ : base B₁) (hB₂ : base B₂) : (B₁ \ B₂).encard = (B₂ \ B₁).encard :=
begin
  obtain (hf | hi) := (B₁ \ B₂).finite_or_infinite,
  { obtain ⟨hf', he⟩ := diff_aux_of_exch exch hB₁ hB₂ hf,
    rw [hf.encard_eq, hf'.encard_eq, he] },
  obtain (hf' | hi') := (B₂ \ B₁).finite_or_infinite,
  { obtain ⟨h, _⟩ := diff_aux_of_exch exch hB₂ hB₁ hf',
    exact (hi h).elim, },
  rw [hi.encard_eq, hi'.encard_eq],
end

private lemma encard_eq_of_exch {base : set α → Prop} (exch :
   exchange_property base)
(hB₁ : base B₁) (hB₂ : base B₂) : B₁.encard = B₂.encard :=
by rw [←encard_diff_add_encard_inter B₁ B₂, encard_diff_eq_of_exch exch
   hB₁ hB₂, inter_comm,
    encard_diff_add_encard_inter]

/-- A `matroid` is a nonempty collection of sets satisfying the exchange
   property and the maximal
  subset property. Each such set is called a `base` of the matroid. -/

@[ext] structure matroid (α : Type*) :=
  (ground : set α)
  (base : set α → Prop)
  (exists_base' : ∃ B, base B)
  (base_exchange' : exchange_property base)
  (maximality : ∀ X ⊆ ground, exists_maximal_subset_property (λ I, ∃ B,
```

```
    base B ∧ I ⊆ B) X)
  (subset_ground' : ∀ B, base B → B ⊆ ground)

end prelim

namespace matroid

section basic -- taken from basic.lean

variables {α : Type*} {I D J B B' B₁ B₂ X Y : set α} {e f : α}

def E (M : matroid α) : set α := M.ground

@[simp] lemma ground_eq_E (M : matroid α) : M.ground = M.E := rfl

section tac

@[user_attribute]
meta def ssE_rules : user_attribute :=
{ name := `ssE_rules,
  descr := "lemmas usable to prove subset of ground set" }

@[user_attribute]
meta def ssE_finish_rules : user_attribute :=
{ name := `ssE_finish_rules,
  descr := "finishing lemmas usable to prove subset of ground set" }

meta def ssE_finish : tactic unit := `[solve_by_elim with ssE_finish_rules
    {max_depth := 2}]

meta def ssE : tactic unit := `[solve_by_elim with ssE_rules
  {max_depth := 3, discharger := ssE_finish}]

@[ssE_rules] private lemma inter_right_subset_ground {X Y : set α} {M :
    matroid α}
(hX : X ⊆ M.E) : X ∩ Y ⊆ M.E := (inter_subset_left _ _).trans hX

@[ssE_rules] private lemma inter_left_subset_ground {X Y : set α} {M :
    matroid α}
(hX : X ⊆ M.E) : Y ∩ X ⊆ M.E := (inter_subset_right _ _).trans hX

@[ssE_rules] private lemma diff_subset_ground {X Y : set α} {M : matroid α
```

```
  }
(hX : X ⊆ M.E) : X \ Y ⊆ M.E := (diff_subset _ _).trans hX

@[simp] lemma ground_inter_right {M : matroid α} (hXE : X ⊆ M.E . ssE) :
  M.E ∩ X = X :=
inter_eq_self_of_subset_right hXE

@[simp] lemma ground_inter_left {M : matroid α} (hXE : X ⊆ M.E . ssE) :
  X ∩ M.E = X :=
inter_eq_self_of_subset_left hXE

@[ssE_rules] private lemma insert_subset_ground {e : α} {X : set α} {M :
  matroid α}
(he : e ∈ M.E) (hX : X ⊆ M.E) : insert e X ⊆ M.E := insert_subset.mpr ⟨
  he, hX⟩

@[ssE_rules] private lemma singleton_subset_ground {e : α} {M : matroid α
  } (he : e ∈ M.E) :
  {e} ⊆ M.E :=
singleton_subset_iff.mpr he

attribute [ssE_rules] mem_of_mem_of_subset empty_subset subset.rfl
  union_subset

end tac

/-- A set is independent if it is contained in a base.  -/
def indep (M : matroid α) (I : set α) : Prop := ∃ B, M.base B ∧ I ⊆ B

/-- A subset of `M.E` is dependent if it is not independent . -/
def dep (M : matroid α) (D : set α) : Prop := ¬M.indep D ∧ D ⊆ M.E

/-- A basis for a set `X ⊆ M.E` is a maximal independent subset of `X`
  (Often in the literature, the word 'basis' is used to refer to what we
   call a 'base'). -/
def basis (M : matroid α) (I X : set α) : Prop :=
  I ∈ maximals (⊆) {A | M.indep A ∧ A ⊆ X} ∧ X ⊆ M.E

/-- A circuit is a minimal dependent set -/
def circuit (M : matroid α) (C : set α) : Prop := C ∈ minimals (⊆) {X |
  M.dep X}
```

```
/-- A coindependent set is a subset of `M.E` that is disjoint from some
   base -/
def coindep (M : matroid α) (I : set α) : Prop := I ⊆ M.E ∧ (∃ B, M.base
   B ∧ disjoint I B)

/-- Typeclass for a matroid having finite ground set. This is just a
   wrapper for `[M.E.finite]`-/
class finite (M : matroid α) : Prop := (ground_finite : M.E.finite)

lemma ground_finite (M : matroid α) [M.finite] : M.E.finite := ⟨
   M.finite⟩.ground_finite

lemma set_finite (M : matroid α) [M.finite] (X : set α) (hX : X ⊆ M.E .
   ssE) : X.finite :=
M.ground_finite.subset hX

instance finite_of_finite [@_root_.finite α] {M : matroid α} : finite M :=
⟨set.to_finite _⟩

class finite_rk (M : matroid α) : Prop := (exists_finite_base : ∃ B,
   M.base B ∧ B.finite)

variables {M : matroid α}

@[ssE_finish_rules] lemma base.subset_ground (hB : M.base B) : B ⊆ M.E :=
M.subset_ground' B hB

lemma exists_base (M : matroid α) : ∃ B, M.base B := M.exists_base'

lemma base.exchange (hB₁ : M.base B₁) (hB₂ : M.base B₂) (hx : e ∈ B₁ \ B₂)
   :
  ∃ y ∈ B₂ \ B₁, M.base (insert y (B₁ \ {e}))  :=
M.base_exchange' B₁ B₂ hB₁ hB₂ _ hx

lemma base.finite_of_finite (hB : M.base B) (h : B.finite) (hB' : M.base
   B') : B'.finite :=
finite_of_finite_of_exch M.base_exchange' hB hB' h

lemma base.finite [finite_rk M] (hB : M.base B) : B.finite :=
let ⟨B₀,hB₀⟩ := ⟨finite_rk M⟩.exists_finite_base in hB₀.1.finite_of_finite
   hB₀.2 hB
```

```
lemma base.finite_rk_of_finite (hB : M.base B) (hfin : B.finite) :
   finite_rk M := ⟨⟨B, hB, hfin⟩⟩

lemma base.encard_eq_encard_of_base (hB₁ : M.base B₁) (hB₂ : M.base B₂) :
  B₁.encard = B₂.encard :=
by rw [encard_eq_of_exch M.base_exchange' hB₁ hB₂]

lemma base.card_eq_card_of_base (hB₁ : M.base B₁) (hB₂ : M.base B₂) :
  B₁.ncard = B₂.ncard :=
by rw [←encard_to_nat_eq B₁, hB₁.encard_eq_encard_of_base hB₂,
   encard_to_nat_eq]

instance finite_rk_of_finite {M : matroid α} [finite M] : finite_rk M :=
let ⟨B, hB⟩ := M.exists_base in ⟨⟨B, hB, (M.ground_finite).subset
   hB.subset_ground⟩⟩

@[ssE_finish_rules] lemma indep.subset_ground (hI : M.indep I) : I ⊆ M.E :
   =
by { obtain ⟨B, hB, hIB⟩ := hI, exact hIB.trans hB.subset_ground }

lemma base.eq_of_subset_base (hB₁ : M.base B₁) (hB₂ : M.base B₂) (hB₁B₂ :
   B₁ ⊆ B₂) :
  B₁ = B₂ :=
antichain_of_exch M.base_exchange' hB₁ hB₂ hB₁B₂

lemma indep_iff_subset_base : M.indep I ↔ ∃ B, M.base B ∧ I ⊆ B :=
   iff.rfl

lemma dep_iff : M.dep D ↔ ¬M.indep D ∧ D ⊆ M.E := iff.rfl

@[ssE_finish_rules] lemma dep.subset_ground (hD : M.dep D) : D ⊆ M.E :=
hD.2

@[ssE_finish_rules] lemma coindep.subset_ground (hX : M.coindep X) : X ⊆
   M.E :=
hX.1

lemma indep.not_dep (hI : M.indep I) : ¬ M.dep I :=
λ h, h.1 hI

lemma dep.not_indep (hD : M.dep D) : ¬ M.indep D :=
hD.1
```

```
lemma dep_of_not_indep (hD : ¬ M.indep D) (hDE : D ⊆ M.E . ssE) : M.dep
   D :=
⟨hD, hDE⟩

lemma indep_of_not_dep (hI : ¬ M.dep I) (hIE : I ⊆ M.E . ssE) : M.indep
   I :=
by_contra (λ h, hI ⟨h, hIE⟩)

@[simp] lemma not_dep_iff (hX : X ⊆ M.E . ssE) : ¬ M.dep X ↔ M.indep X :
   =
by rw [dep, and_iff_left hX, not_not]

@[simp] lemma not_indep_iff (hX : X ⊆ M.E . ssE) : ¬ M.indep X ↔ M.dep
   X :=
by rw [dep, and_iff_left hX]

lemma indep.exists_base_supset (hI : M.indep I) : ∃ B, M.base B ∧ I ⊆ B :
   =
hI

lemma indep.subset (hJ : M.indep J) (hIJ : I ⊆ J) : M.indep I :=
by {obtain ⟨B, hB, hJB⟩ := hJ, exact ⟨B, hB, hIJ.trans hJB⟩}

lemma dep.supset (hD : M.dep D) (hDX : D ⊆ X) (hXE : X ⊆ M.E . ssE) :
   M.dep X :=
dep_of_not_indep (λ hI, (hI.subset hDX).not_dep hD)

@[simp] lemma empty_indep (M : matroid α) : M.indep ∅ :=
exists.elim M.exists_base (λ B hB, ⟨_, hB, B.empty_subset⟩)

lemma indep.finite [finite_rk M] (hI : M.indep I) : I.finite :=
let ⟨B, hB, hIB⟩ := hI in hB.finite.subset hIB

lemma indep.inter_right (hI : M.indep I) (X : set α) : M.indep (I ∩ X) :=
hI.subset (inter_subset_left _ _)

lemma indep.diff (hI : M.indep I) (X : set α) : M.indep (I \ X) :=
   hI.subset (diff_subset _ _)

lemma base.indep (hB : M.base B) : M.indep B := ⟨B, hB, subset_rfl⟩
```

```
lemma base.eq_of_subset_indep (hB : M.base B) (hI : M.indep I) (hBI : B ⊆
    I) : B = I :=
let ⟨B', hB', hB'I⟩ := hI in hBI.antisymm (by rwa hB.eq_of_subset_base hB'
    (hBI.trans hB'I))

lemma base_iff_maximal_indep : M.base B ↔ M.indep B ∧ ∀ I, M.indep I →
    B ⊆ I → B = I :=
begin
  refine ⟨λ h, ⟨h.indep, λ _, h.eq_of_subset_indep ⟩,λ h, _⟩,
  obtain ⟨⟨B', hB', hBB'⟩, h⟩ := h,
  rwa h _ hB'.indep hBB',
end

lemma base_iff_mem_maximals : M.base B ↔ B ∈ maximals (⊆) {I | M.indep
    I} :=
begin
  rw [base_iff_maximal_indep, maximals],
  exact ⟨λ h, ⟨h.1,λ I hI hBI, (h.2 I hI hBI).symm.subset⟩,
    λ h, ⟨h.1, λ I hI hBI, hBI.antisymm (h.2 hI hBI)⟩⟩,
end

lemma indep.base_of_maximal (hI : M.indep I) (h : ∀ J, M.indep J → I ⊆ J
    → I = J) : M.base I :=
base_iff_maximal_indep.mpr ⟨hI,h⟩

/-- If the difference of two bases is a singleton, then they differ by
    an insertion/removal -/
lemma base.eq_exchange_of_diff_eq_singleton (hB : M.base B) (hB' : M.base
    B') (h : B \ B' = {e}) :
  ∃ f ∈ B' \ B, B' = (insert f B) \ {e} :=
begin
  obtain ⟨f, hf, hb⟩ := hB.exchange hB' (h.symm.subset (mem_singleton e)),
  have hne : f ≠ e,
  { rintro rfl, exact hf.2 (h.symm.subset (mem_singleton f)).1 },
  rw insert_diff_singleton_comm hne at hb,
  refine ⟨f, hf, (hb.eq_of_subset_base hB' _).symm⟩,
  rw [diff_subset_iff, insert_subset, union_comm, ←diff_subset_iff, h,
    and_iff_left subset.rfl],
  exact or.inl hf.1,
end

lemma basis.indep (hI : M.basis I X) : M.indep I := hI.1.1.1
```

```
lemma basis.subset (hI : M.basis I X) : I ⊆ X := hI.1.1.2

@[ssE_finish_rules] lemma basis.subset_ground (hI : M.basis I X) : X ⊆
   M.E :=
hI.2

lemma basis.basis_inter_ground (hI : M.basis I X) : M.basis I (X ∩ M.E) :=
by { convert hI, rw [inter_eq_self_of_subset_left hI.subset_ground] }

@[ssE_finish_rules] lemma basis.subset_ground_left (hI : M.basis I X) : I
   ⊆ M.E :=
hI.indep.subset_ground

lemma basis.eq_of_subset_indep (hI : M.basis I X) (hJ : M.indep J) (hIJ :
   I ⊆ J) (hJX : J ⊆ X) :
  I = J :=
hIJ.antisymm (hI.1.2 ⟨hJ, hJX⟩ hIJ)

lemma basis.finite (hI : M.basis I X) [finite_rk M] : I.finite :=
   hI.indep.finite

lemma basis_iff' :
  M.basis I X ↔ (M.indep I ∧ I ⊆ X ∧ ∀ J, M.indep J → I ⊆ J → J ⊆ X
    → I = J) ∧ X ⊆ M.E :=
begin
  simp_rw [basis, and.congr_left_iff, maximals, mem_set_of_eq, and_imp,
   sep_set_of,
    mem_set_of_eq, and_assoc, and.congr_right_iff],
  intros hXE hI hIX,
  exact ⟨λ h J hJ hIJ hJX, hIJ.antisymm (h hJ hJX hIJ),
    λ h J hJ hIJ hJX, (h J hJ hJX hIJ).symm.subset⟩,
end

lemma basis_iff (hX : X ⊆ M.E . ssE) :
  M.basis I X ↔ (M.indep I ∧ I ⊆ X ∧ ∀ J, M.indep J → I ⊆ J → J ⊆ X
    → I = J) :=
by rw [basis_iff', and_iff_left hX]

lemma basis_iff_mem_maximals (hX : X ⊆ M.E . ssE):
  M.basis I X ↔ I ∈ maximals (⊆) (λ (I : set α), M.indep I ∧ I ⊆ X) :=
begin
```

```
  simp_rw [basis_iff, mem_maximals_prop_iff, and_assoc, and_imp,
   and.congr_right_iff],
  exact λ hI hIX, ⟨λ h J hJ hJX hIJ, h J hJ hIJ hJX, λ h J hJ hJX hIJ, h
   hJ hIJ hJX⟩,
end

lemma basis.basis_subset (hI : M.basis I X) (hIY : I ⊆ Y) (hYX : Y ⊆ X) :
    M.basis I Y :=
begin
  rw [basis_iff (hYX.trans hI.subset_ground), and_iff_right hI.indep,
   and_iff_right hIY],
  exact λ J hJ hIJ hJY, hI.eq_of_subset_indep hJ hIJ (hJY.trans hYX),
end

lemma basis.dep_of_ssubset (hI : M.basis I X) {Y : set α} (hIY : I ⊂ Y)
   (hYX : Y ⊆ X) : M.dep Y :=
begin
  rw [←not_indep_iff (hYX.trans hI.subset_ground)],
  exact λ hY, hIY.ne (hI.eq_of_subset_indep hY hIY.subset hYX),
end

lemma basis.insert_dep (hI : M.basis I X) (he : e ∈ X \ I) : M.dep
   (insert e I) :=
hI.dep_of_ssubset (ssubset_insert he.2) (insert_subset.mpr ⟨
   he.1,hI.subset⟩)

lemma basis.mem_of_insert_indep (hI : M.basis I X) (he : e ∈ X) (hIe :
   M.indep (insert e I)) :
  e ∈ I :=
by_contra (λ heI, (hI.insert_dep ⟨he, heI⟩).not_indep hIe)

lemma indep.subset_basis_of_subset (hI : M.indep I) (hIX : I ⊆ X) (hX : X
   ⊆ M.E . ssE) :
  ∃ J, M.basis J X ∧ I ⊆ J :=
begin
  obtain ⟨J, ⟨(hJ : M.indep J),hIJ,hJX⟩, hJmax⟩ := M.maximality X hX I hI
   hIX,
  use J,
  rw [and_iff_left hIJ, basis_iff, and_iff_right hJ, and_iff_right hJX],
  exact λ K hK hJK hKX, hJK.antisymm (hJmax ⟨hK, hIJ.trans hJK, hKX⟩ hJK),
end
```

```
lemma exists_basis (M : matroid α) (X : set α) (hX : X ⊆ M.E . ssE) : ∃
    I, M.basis I X :=
let ⟨I, hI, _⟩ := M.empty_indep.subset_basis_of_subset (empty_subset X) in
    ⟨_,hI⟩

lemma basis.exists_basis_inter_eq_of_supset (hI : M.basis I X) (hXY : X ⊆
    Y) (hY : Y ⊆ M.E . ssE) :
  ∃ J, M.basis J Y ∧ J ∩ X = I :=
begin
  obtain ⟨J, hJ, hIJ⟩ := hI.indep.subset_basis_of_subset (hI.subset.trans
    hXY),
  refine ⟨J, hJ, subset_antisymm _ (subset_inter hIJ hI.subset)⟩,
  exact λ e he, hI.mem_of_insert_indep he.2 (hJ.indep.subset
    (insert_subset.mpr ⟨he.1, hIJ⟩)),
end

lemma indep.exists_insert_of_not_base (hI : M.indep I) (hI' : ¬M.base I)
    (hB : M.base B) :
  ∃ e ∈ B \ I, M.indep (insert e I) :=
begin
  obtain ⟨B', hB', hIB'⟩ := hI,
  obtain ⟨x, hxB', hx⟩ := exists_of_ssubset (hIB'.ssubset_of_ne (by {
    rintro rfl, exact hI' hB' })),
  obtain (hxB | hxB) := em (x ∈ B),
  { exact ⟨x, ⟨hxB, hx⟩ , ⟨B', hB', insert_subset.mpr ⟨hxB',hIB'⟩⟩⟩ },
  obtain ⟨e,he, hbase⟩ := hB'.exchange hB ⟨hxB',hxB⟩,
  exact ⟨e, ⟨he.1, not_mem_subset hIB' he.2⟩,
    ⟨_, hbase, insert_subset_insert (subset_diff_singleton hIB' hx)⟩⟩,
end

lemma indep.basis_self (hI : M.indep I) : M.basis I I :=
begin
  rw [basis_iff', and_iff_left hI.subset_ground, and_iff_right hI,
    and_iff_right subset.rfl],
  exact λ _ _, subset_antisymm,
end

lemma basis.exists_base (hI : M.basis I X) : ∃ B, M.base B ∧ I = B ∩ X :=
begin
  obtain ⟨B,hB, hIB⟩ := hI.indep,
  refine ⟨B, hB, subset_antisymm (subset_inter hIB hI.subset) _⟩,
  rw hI.eq_of_subset_indep (hB.indep.inter_right X) (subset_inter hIB
```

```
   hI.subset)
    (inter_subset_right _ _),
end

lemma base_iff_basis_ground : M.base B ↔ M.basis B M.E :=
begin
  rw [base_iff_maximal_indep, basis_iff, and_congr_right],
  intro hB,
  rw [and_iff_right hB.subset_ground],
  exact ⟨λ h J hJ hBJ hJE, h _ hJ hBJ, λ h I hI hBI, h I hI hBI
   hI.subset_ground⟩,
end

lemma base.basis_ground (hB : M.base B) : M.basis B M.E :=
   base_iff_basis_ground.mp hB

lemma indep.basis_of_forall_insert (hI : M.indep I)
  (hIX : I ⊆ X) (he : ∀ e ∈ X \ I, ¬ M.indep (insert e I)) (hX : X ⊆
   M.E . ssE) : M.basis I X :=
begin
  rw [basis_iff, and_iff_right hI, and_iff_right hIX],
  refine λJ hJ hIJ hJX, hIJ.antisymm (λ e heJ, by_contra (λ heI, he e ⟨
   hJX heJ, heI⟩ _)),
  exact hJ.subset (insert_subset.mpr ⟨heJ, hIJ⟩),
end

lemma basis.Union_basis_Union {ι : Type*} (X I : ι → set α) (hI : ∀ i,
   M.basis (I i) (X i))
(h_ind : M.indep (⋃ i, I i)) : M.basis (⋃ i, I i) (⋃ i, X i) :=
begin

  refine h_ind.basis_of_forall_insert
    (Union_subset_iff.mpr (λ i, (hI i).subset.trans (subset_Union _ _)))
   (λ e he hi, _)
    (Union_subset (λ i, (hI i).subset_ground)),
  simp only [mem_diff, mem_Union, not_exists] at he,
  obtain ⟨i, heXi⟩ := he.1,
  exact he.2 i ((hI i).mem_of_insert_indep heXi
    (hi.subset (insert_subset_insert (subset_Union _ _)))),
end

lemma basis.union_basis_union (hIX : M.basis I X) (hJY : M.basis J Y) (h :
```

```
    M.indep (I ∪ J)) :
  M.basis (I ∪ J) (X ∪ Y) :=
begin
  rw [union_eq_Union, union_eq_Union],
  refine basis.Union_basis_Union _ _ _ _,
  { simp only [bool.forall_bool, bool.cond_ff, bool.cond_tt], exact ⟨hJY,
   hIX⟩ },
  rwa [←union_eq_Union],
end

lemma basis.basis_union (hIX : M.basis I X) (hIY : M.basis I Y) : M.basis
   I (X ∪ Y) :=
by { convert hIX.union_basis_union hIY _; rw union_self, exact hIX.indep }

lemma basis.basis_union_of_subset (hI : M.basis I X) (hJ : M.indep J)
   (hIJ : I ⊆ J) :
  M.basis J (J ∪ X) :=
begin
  convert hJ.basis_self.union_basis_union hI (by rwa
   union_eq_left_iff_subset.mpr hIJ),
  rw union_eq_left_iff_subset.mpr hIJ,
end

lemma basis.insert_basis_insert (hI : M.basis I X) (h : M.indep (insert e
   I)) :
  M.basis (insert e I) (insert e X) :=
begin
  convert hI.union_basis_union
    (indep.basis_self (h.subset (singleton_subset_iff.mpr (mem_insert _
   _)))) _,
  simp, simp, simpa,
end

lemma base.base_of_basis_supset (hB : M.base B) (hBX : B ⊆ X) (hIX :
   M.basis I X) : M.base I :=
begin
  by_contra h,
  obtain ⟨e,heBI,he⟩ := hIX.indep.exists_insert_of_not_base h hB,
  exact heBI.2 (hIX.mem_of_insert_indep (hBX heBI.1) he),
end

lemma base.basis_of_subset (hX : X ⊆ M.E . ssE) (hB : M.base B) (hBX : B
```

```
   ⊆ X) : M.basis B X :=
begin
  rw [basis_iff, and_iff_right hB.indep, and_iff_right hBX],
  exact λ J hJ hBJ hJX, hB.eq_of_subset_indep hJ hBJ,
end

lemma indep.exists_base_subset_union_base (hI : M.indep I) (hB : M.base
   B) :
  ∃ B', M.base B' ∧ I ⊆ B' ∧ B' ⊆ I ∪ B :=
begin
  obtain ⟨B', hB', hIB'⟩ := hI.subset_basis_of_subset (subset_union_left I
   B),
  exact ⟨B', hB.base_of_basis_supset (subset_union_right _ _) hB', hIB',
   hB'.subset⟩,
end

lemma eq_of_base_iff_base_forall {M₁ M₂ : matroid α} (hE : M₁.E = M₂.E)
(h : ∀ B ⊆ M₁.E, (M₁.base B ↔ M₂.base B)) : M₁ = M₂ :=
begin
  apply matroid.ext _ _ hE,
  ext B,
  refine ⟨λ h', (h _ h'.subset_ground).mp h',
    λ h', (h _ (h'.subset_ground.trans_eq hE.symm)).mpr h'⟩,
end

lemma eq_of_indep_iff_indep_forall {M₁ M₂ : matroid α} (hE : M₁.E = M₂.E)
(h : ∀ I ⊆ M₁.E, (M₁.indep I ↔ M₂.indep I)) :
  M₁ = M₂ :=
begin
  refine eq_of_base_iff_base_forall hE (λ B hB, _),
  rw [base_iff_maximal_indep, base_iff_maximal_indep],
  split,
  { rintro ⟨hBi, hmax⟩,
    rw [←h _ hB, and_iff_right hBi],
    refine λ I hI hBI, hmax I _ hBI,
    rwa h,
    rw [hE],
    exact hI.subset_ground },
  rintro ⟨hBi, hmax⟩,
  rw [h _ hB, and_iff_right hBi],
  refine λ I hI hBI,  hmax I _ hBI,
  rwa ←h,
```

```
  exact hI.subset_ground,
end

lemma eq_iff_indep_iff_indep_forall {M₁ M₂ : matroid α} :
  M₁ = M₂ ↔ (M₁.E = M₂.E) ∧ ∀ I ⊆ M₁.E , M₁.indep I ↔ M₂.indep I :=
⟨λ h, by { subst h, simp }, λ h, eq_of_indep_iff_indep_forall h.1 h.2⟩

def matroid_of_base {α : Type*} (E : set α) (base : set α → Prop)
(exists_base : ∃ B, base B) (base_exchange : exchange_property base)
(maximality : ∀ X ⊆ E, exists_maximal_subset_property (λ I, ∃ B, base B ∧
    I ⊆ B) X)
(support : ∀ B, base B → B ⊆ E) : matroid α :=
⟨E, base, exists_base, base_exchange, maximality, support⟩

@[simp] lemma matroid_of_base_apply {α : Type*} (E : set α) (base : set α
   → Prop)
(exists_base : ∃ B, base B) (base_exchange : exchange_property base)
(maximality : ∀ X ⊆ E, exists_maximal_subset_property (λ I, ∃ B, base B ∧
    I ⊆ B) X)
(support : ∀ B, base B → B ⊆ E) :
(matroid_of_base E base exists_base base_exchange maximality
   support).base = base := rfl

/-- A version of the independence axioms that avoids cardinality -/
def matroid_of_indep (E : set α) (indep : set α → Prop) (h_empty : indep
   ∅)
(h_subset : ∀ {I J}, indep J → I ⊆ J → indep I)
(h_aug : ∀{I B}, indep I → I ∉ maximals (⊆) indep → B ∈ maximals (⊆)
   indep →
  ∃ x ∈ B \ I, indep (insert x I))
(h_maximal : ∀ X ⊆ E, exists_maximal_subset_property indep X)
(h_support : ∀ I, indep I → I ⊆ E) :
  matroid α :=
matroid_of_base E (λ B, B ∈ maximals (⊆) indep)
(begin
  obtain ⟨B, ⟨hB,-,-⟩, hB₁⟩ := h_maximal E subset.rfl ∅ h_empty
    (empty_subset _),
  exact ⟨B, hB, λ B' hB' hBB', hB₁ ⟨hB',empty_subset _,h_support B' hB'⟩ hBB'⟩,
end)
(begin
  rintros B B' ⟨hB,hBmax⟩ ⟨hB',hB'max⟩ e he,
  obtain ⟨f,hf,hfB⟩ :=  h_aug (h_subset hB (diff_subset B {e})) _ ⟨
```

```
   hB',hB'max⟩,
  simp only [mem_diff, mem_singleton_iff, not_and, not_not] at hf,
  have hfB' : f ∉ B,
  { intro hfB, have := hf.2 hfB, subst this, exact he.2 hf.1 },
  { refine ⟨f, ⟨hf.1, hfB'⟩, by_contra (λ hnot, _)⟩,
    obtain ⟨x,hxB, hind⟩ :=  h_aug hfB hnot ⟨hB, hBmax⟩,
    simp only [mem_diff, mem_insert_iff, mem_singleton_iff,
   not_or_distrib, not_and, not_not]
      at hxB,
    have := hxB.2.2 hxB.1, subst this,
    rw [insert_comm, insert_diff_singleton, insert_eq_of_mem he.1] at
   hind,
    exact not_mem_subset (hBmax hind (subset_insert _ _)) hfB' (mem_insert
   _ _) },
  simp only [maximals, mem_sep_iff, diff_singleton_subset_iff, not_and,
   not_forall, exists_prop],
  exact λ _, ⟨B, hB, subset_insert _ _, λ hss, (hss he.1).2 rfl⟩,
end)
(begin
  rintro X hXE I ⟨B, hB, hIB⟩ hIX,
  -- rintro I X ⟨B, hB,  hIB⟩ hIX,
  obtain ⟨J, ⟨hJ, hIJ, hJX⟩, hJmax⟩ := h_maximal X hXE I (h_subset hB.1
   hIB) hIX,
  obtain ⟨BJ, hBJ⟩ := h_maximal E subset.rfl J hJ (h_support J hJ),
  refine ⟨J, ⟨⟨BJ,_, hBJ.1.2.1⟩ ,hIJ,hJX⟩, _⟩,
  { exact ⟨hBJ.1.1, λ B' hB' hBJB', hBJ.2 ⟨hB',hBJ.1.2.1.trans hBJB',
   h_support B' hB'⟩ hBJB'⟩ },
  simp only [mem_set_of_eq, and_imp, forall_exists_index],
  rintro A B' ⟨(hB'i : indep _), hB'max⟩ hB'' hIA hAX hJA,
  exact hJmax ⟨h_subset hB'i hB'', hIA, hAX⟩ hJA,
end )
(λ B hB, h_support B hB.1)

@[simp] lemma matroid_of_indep_apply (E : set α) (indep : set α → Prop)
   (h_empty : indep ∅)
(h_subset : ∀ {I J}, indep J → I ⊆ J → indep I)
(h_aug : ∀{I B}, indep I → I ∉ maximals (⊆) indep → B ∈ maximals (⊆)
   indep →
  ∃ x ∈ B \ I, indep (insert x I))
(h_maximal : ∀ X ⊆ E, exists_maximal_subset_property indep X)
(h_support : ∀ I, indep I → I ⊆ E)  :
(matroid_of_indep E indep h_empty h_subset h_aug h_maximal
```

```
   h_support).indep = indep :=
begin
  ext I,
  simp only [matroid.indep, matroid_of_indep],
  refine ⟨λ ⟨B, hB, hIB⟩, h_subset hB.1 hIB, λ hI, _⟩,
  obtain ⟨B, ⟨hB, hIB, -⟩, hBmax⟩ :=  h_maximal E subset.rfl I hI
   (h_support _ hI),
  exact ⟨B, ⟨hB, λ B' hB' hBB', hBmax ⟨hB', hIB.trans hBB', h_support _ hB'⟩
   hBB'⟩, hIB⟩,
end

/-- If there is an absolute upper bound on the size of an independent
   set, then the maximality
  axiom isn't needed to define a matroid by independent sets. -/
def matroid_of_indep_of_bdd (E : set α) (indep : set α → Prop) (h_empty :
    indep ∅)
(h_subset : ∀ {I J}, indep J → I ⊆ J → indep I)
(h_aug : ∀{I B}, indep I → I ∉ maximals (⊆) indep → B ∈ maximals (⊆)
   indep →
  ∃ x ∈ B \ I, indep (insert x I))
(h_bdd : ∃ n, ∀ I, indep I → I.finite ∧ I.ncard ≤ n )
(h_support : ∀ I, indep I → I ⊆ E) : matroid α :=
matroid_of_indep E indep h_empty h_subset h_aug
  (λ X h, exists_maximal_subset_property_of_bounded h_bdd X)
  h_support

@[simp] lemma matroid_of_indep_of_bdd_apply (E : set α) (indep : set α →
   Prop) (h_empty : indep ∅)
(h_subset : ∀ {I J}, indep J → I ⊆ J → indep I)
(h_aug : ∀{I B}, indep I → I ∉ maximals (⊆) indep → B ∈ maximals (⊆)
   indep →
  ∃ x ∈ B \ I, indep (insert x I))
(h_bdd : ∃ n, ∀ I, indep I → I.finite ∧ I.ncard ≤ n )
(h_support : ∀ I, indep I → I ⊆ E) :
(matroid_of_indep_of_bdd E indep h_empty h_subset h_aug h_bdd
   h_support).indep = indep :=
by simp [matroid_of_indep_of_bdd]

instance (E : set α) (indep : set α → Prop) (h_empty : indep ∅)
(h_subset : ∀ {I J}, indep J → I ⊆ J → indep I)
(h_aug : ∀{I B}, indep I → I ∉ maximals (⊆) indep → B ∈ maximals (⊆)
   indep →
```

```
  ∃ x ∈ B \ I, indep (insert x I)) (h_bdd : ∃ n, ∀ I, indep I → I.finite
   ∧ I.ncard ≤ n )
(h_support : ∀ I, indep I → I ⊆ E) :
matroid.finite_rk (matroid_of_indep_of_bdd E indep h_empty h_subset h_aug
   h_bdd h_support) :=
begin
  obtain ⟨B, hB⟩ :=
    (matroid_of_indep_of_bdd E indep h_empty h_subset h_aug h_bdd
   h_support).exists_base,
  obtain ⟨h, h_bdd⟩ := h_bdd,
  refine hB.finite_rk_of_finite (h_bdd B _).1,
  rw [←matroid_of_indep_of_bdd_apply E indep, matroid.indep],
  exact ⟨_, hB, subset.rfl⟩,
end

def matroid_of_indep_of_bdd' (E : set α) (indep : set α → Prop) (h_empty
   : indep ∅)
(h_subset : ∀ {I J}, indep J → I ⊆ J → indep I)
(ind_aug : ∀ {I J}, indep I → indep J → I.ncard < J.ncard →
  ∃ e ∈ J, e ∉ I ∧ indep (insert e I)) (h_bdd : ∃ n, ∀ I, indep I →
   I.finite ∧ I.ncard ≤ n )
(h_support : ∀ I, indep I → I ⊆ E) : matroid α :=
matroid_of_indep_of_bdd E indep h_empty h_subset
(begin
  intros I J hI hIn hJ,
  by_contra' h',
  obtain (hlt | hle) := lt_or_le I.ncard J.ncard,
  { obtain ⟨e,heJ,heI, hi⟩ :=  ind_aug hI hJ.1 hlt,
    exact h' e ⟨heJ,heI⟩ hi },
  obtain (h_eq | hlt) := hle.eq_or_lt,
  { refine hIn ⟨hI, λ K (hK : indep K) hIK, hIK.ssubset_or_eq.elim (λ
   hss, _)
      (λ h, h.symm.subset)⟩,
    obtain ⟨f, hfK, hfJ, hi⟩ := ind_aug hJ.1 hK (h_eq.trans_lt
   (ncard_lt_ncard hss _)),
    { exact (hfJ (hJ.2 hi (subset_insert _ _) (mem_insert f _))).elim },
    obtain ⟨n, hn⟩ := h_bdd,
    exact (hn K hK).1 },
  obtain ⟨e,heI, heJ, hi⟩ := ind_aug hJ.1 hI hlt,
    exact heJ (hJ.2 hi (subset_insert _ _) (mem_insert e _)),
end) h_bdd h_support
```

```
@[simp] lemma matroid_of_indep_of_bdd'_apply (E : set α) (indep : set α →
    Prop) (h_empty : indep ∅)
(h_subset : ∀ {⦃I J⦄}, indep J → I ⊆ J → indep I)
(ind_aug : ∀ {⦃I J⦄}, indep I → indep J → I.ncard < J.ncard →
  ∃ e ∈ J, e ∉ I ∧ indep (insert e I)) (h_bdd : ∃ n, ∀ I, indep I →
   I.finite ∧ I.ncard ≤ n )
(h_support : ∀ I, indep I → I ⊆ E) :
(matroid_of_indep_of_bdd' E indep h_empty h_subset ind_aug h_bdd
   h_support).indep = indep :=
by simp [matroid_of_indep_of_bdd']

lemma base_compl_iff_mem_maximals_disjoint_base' (hB : B ⊆ M.E . ssE) :
  M.base (M.E \ B) ↔ B ∈ maximals (⊆) {I | I ⊆ M.E ∧ ∃ B, M.base B ∧
   disjoint I B} :=
begin
  refine ⟨λ h, ⟨⟨hB,_,h,disjoint_sdiff_right⟩,_⟩, λ h, _⟩,
  { rintro X ⟨hXE,B', hB', hXB'⟩ hBX,
    rw [hB'.eq_of_subset_base h (subset_diff.mpr ⟨hB'.subset_ground,_⟩),
      ←subset_compl_iff_disjoint_right, diff_eq, compl_inter,
   compl_compl] at hXB',
    { refine (subset_inter hXE hXB').trans _,
      rw [inter_distrib_left, inter_compl_self, empty_union],
      exact inter_subset_right _ _ },
    exact (disjoint_of_subset_left hBX hXB').symm },
  obtain ⟨⟨-, B', hB', hIB'⟩, h⟩ := h,
  suffices : B' = M.E \ B, rwa ←this,
  rw [subset_antisymm_iff, subset_diff, disjoint.comm, and_iff_left hIB',
    and_iff_right hB'.subset_ground, diff_subset_iff],

  intros e he,
  rw [mem_union, or_iff_not_imp_right],
  intros heB',
  refine h ⟨insert_subset.mpr ⟨he, hB⟩, ⟨B', hB', _⟩⟩
    (subset_insert _ _) (mem_insert e B),
  rw [←union_singleton, disjoint_union_left, disjoint_singleton_left],
  exact ⟨hIB', heB'⟩,
end

def dual (M : matroid α) : matroid α :=
matroid_of_indep M.E (λ I, I ⊆ M.E ∧ ∃ B, M.base B ∧ disjoint I B)
⟨empty_subset M.E, M.exists_base.imp (λ B hB, ⟨hB, empty_disjoint _⟩)⟩
(begin
```

```
  rintro I J ⟨hJE, B, hB, hJB⟩ hIJ,
  exact ⟨hIJ.trans hJE, ⟨B, hB, disjoint_of_subset_left hIJ hJB⟩⟩,
end)
(begin
  rintro I X ⟨hIE, B, hB, hIB⟩ hI_not_max hX_max,
  have hXE := hX_max.1.1,
  have hB' := (base_compl_iff_mem_maximals_disjoint_base' hXE).mpr hX_max,

  set B' := M.E \ X with hX,
  have hI := (not_iff_not.mpr
   (base_compl_iff_mem_maximals_disjoint_base')).mpr hI_not_max,
  obtain ⟨B'', hB'', hB''₁, hB''₂⟩ := (hB'.indep.diff
   I).exists_base_subset_union_base hB,
  rw [←compl_subset_compl, ←hIB.sdiff_eq_right, ←union_diff_distrib,
   diff_eq, compl_inter,
    compl_compl, union_subset_iff, compl_subset_compl] at hB''₂,

  have hssu := (subset_inter (hB''₂.2) hIE).ssubset_of_ne
    (by { rintro rfl, apply hI, convert hB'', simp }),
  obtain ⟨e, ⟨(heB'' : e ∉ _), heE⟩, heI⟩ := exists_of_ssubset hssu,
  use e,
  rw [mem_diff, insert_subset, and_iff_left heI, and_iff_right heE,
   and_iff_right hIE],
  refine ⟨by_contra (λ heX, heB'' (hB''₁ ⟨_, heI⟩)), ⟨B'', hB'', _⟩⟩,
  { rw [hX], exact ⟨heE, heX⟩ },
  rw [←union_singleton, disjoint_union_left, disjoint_singleton_left,
   and_iff_left heB''],
  exact disjoint_of_subset_left hB''₂.2 disjoint_compl_left,
end)
(begin
  rintro X hX I' ⟨hI'E, B, hB, hI'B⟩ hI'X,
  obtain ⟨I, hI⟩ :=  M.exists_basis (M.E \ X) ,
  obtain ⟨B', hB', hIB', hB'IB⟩ := hI.indep.exists_base_subset_union_base hB,
  refine ⟨(X \ B') ∩ M.E,
    ⟨_,subset_inter (subset_diff.mpr _) hI'E, (inter_subset_left _
    _).trans (diff_subset _ _)⟩, _⟩,
  { simp only [inter_subset_right, true_and],
    exact ⟨B', hB', disjoint_of_subset_left (inter_subset_left _ _)
   disjoint_sdiff_left⟩ },
  { rw [and_iff_right hI'X],
    refine disjoint_of_subset_right hB'IB _,
    rw [disjoint_union_right, and_iff_left hI'B],
```

```
    exact disjoint_of_subset hI'X hI.subset disjoint_sdiff_right },
  simp only [mem_set_of_eq, subset_inter_iff, and_imp,
   forall_exists_index],
  intros J hJE B'' hB'' hdj hI'J hJX hssJ,
  rw [and_iff_left hJE],
  rw [diff_eq, inter_right_comm, ←diff_eq, diff_subset_iff] at hssJ,

  have hI' : (B'' ∩ X) ∪ (B' \ X) ⊆ B',
  { rw [union_subset_iff, and_iff_left (diff_subset _ _),
     ←inter_eq_self_of_subset_left hB''.subset_ground, inter_right_comm,
    inter_assoc],
    calc _ ⊆ _ : inter_subset_inter_right _ hssJ
       ... ⊆ _ : by rw [inter_distrib_left, hdj.symm.inter_eq,
    union_empty]
       ... ⊆ _ : inter_subset_right _ _ },

  obtain ⟨B₁,hB₁,hI'B₁,hB₁I⟩ := (hB'.indep.subset
   hI').exists_base_subset_union_base hB'',
  rw [union_comm, ←union_assoc, union_eq_self_of_subset_right
   (inter_subset_left _ _)] at hB₁I,

  have : B₁ = B',
  { refine hB₁.eq_of_subset_indep hB'.indep (λ e he, _),
    refine (hB₁I he).elim (λ heB'',_) (λ h, h.1),
    refine (em (e ∈ X)).elim (λ heX, hI' (or.inl ⟨heB'', heX⟩)) (λ heX,
    hIB' _),
    refine hI.mem_of_insert_indep ⟨hB₁.subset_ground he, heX⟩
      (hB₁.indep.subset (insert_subset.mpr ⟨he, _⟩)),
    refine (subset_union_of_subset_right (subset_diff.mpr ⟨hIB',_⟩)
    _).trans hI'B₁,
    refine disjoint_of_subset_left hI.subset disjoint_sdiff_left },

  subst this,

  refine subset_diff.mpr ⟨hJX, by_contra (λ hne, _)⟩,
  obtain ⟨e, heJ, heB'⟩ := not_disjoint_iff.mp hne,
  obtain (heB'' | ⟨heB₁,heX⟩ ) := hB₁I heB',
  { exact hdj.ne_of_mem heJ heB'' rfl },
  exact heX (hJX heJ),
end)
(by tauto)
```

```
/-- A notation typeclass for matroid duality, denoted by the `*` symbol.
   -/
@[class] structure has_matroid_dual (β : Type*) := (dual : β → β)

postfix `*`:(max+1) := has_matroid_dual.dual

instance matroid_dual {α : Type*} : has_matroid_dual (matroid α) := ⟨
   matroid.dual⟩

lemma dual_indep_iff_exists' : (M*.indep I) ↔ I ⊆ M.E ∧ (∃ B, M.base B ∧
   disjoint I B) :=
by simp [has_matroid_dual.dual, dual]

@[simp] lemma dual_ground : M*.E = M.E := rfl

lemma dual_base_iff (hB : B ⊆ M.E . ssE) : M*.base B ↔ M.base (M.E \ B) :
   =
begin
  rw [base_compl_iff_mem_maximals_disjoint_base', base_iff_maximal_indep,
   dual_indep_iff_exists',
    mem_maximals_set_of_iff],
  simp [dual_indep_iff_exists'],
end

lemma dual_base_iff' : M*.base B ↔ M.base (M.E \ B) ∧ B ⊆ M.E :=
begin
  obtain (h | h) := em (B ⊆ M.E),
  { rw [dual_base_iff, and_iff_left h] },
  rw [iff_false_intro h, and_false, iff_false],
  exact λ h', h h'.subset_ground,
end

@[simp] lemma dual_dual (M : matroid α) : M** = M :=
begin
  refine eq_of_base_iff_base_forall rfl (λ B hB, _),
  rw [dual_base_iff, dual_base_iff],
  rw [dual_ground] at *,
  simp only [sdiff_sdiff_right_self, inf_eq_inter, ground_inter_right],
end

lemma dual_indep_iff_coindep : M*.indep X ↔ M.coindep X :=
   dual_indep_iff_exists'
```

```
lemma base.compl_base_dual (hB : M.base B) : M✶.base (M.E \ B) :=
by { haveI := fact.mk hB.subset_ground, simpa [dual_base_iff] }

lemma base.compl_inter_basis_of_inter_basis (hB : M.base B) (hBX :
   M.basis (B ∩ X) X) :
  M✶.basis ((M.E \ B) ∩ (M.E \ X)) (M.E \ X) :=
begin
  rw basis_iff,
  refine ⟨(hB.compl_base_dual.indep.subset (inter_subset_left _ _)),
   inter_subset_right _ _,
    λ J hJ hBCJ hJX, hBCJ.antisymm (subset_inter _ hJX)⟩,

  obtain ⟨-, B', hB', hJB'⟩ := dual_indep_iff_coindep.mp hJ,

  obtain ⟨B'', hB'', hsB'', hB''s⟩  := hBX.indep.exists_base_subset_union_base
   hB',
  have hB'ss : B' ⊆ B ∪ X,
  { rw [←diff_subset_diff_iff M.E (by ssE : B ∪ X ⊆ M.E)
   hB'.subset_ground, subset_diff,
      and_iff_right (diff_subset _ _)],
    rw [diff_inter_diff] at hBCJ,
    exact disjoint_of_subset_left hBCJ hJB' },

  have hB''ss : B'' ⊆ B,
  { refine λ e he, (hB''s he).elim and.left (λ heB', (hB'ss heB').elim id (λ
   heX, _)),
     exact (hBX.mem_of_insert_indep heX (hB''.indep.subset
   (insert_subset.mpr ⟨he,hsB''⟩))).1 },

  have := (hB''.eq_of_subset_indep hB.indep hB''ss).symm, subst this,
  rw subset_diff at *,
  exact ⟨hJX.1, disjoint_of_subset_right hB''s (disjoint_union_right.mpr
    ⟨disjoint_of_subset_right (inter_subset_right _ _) hJX.2, hJB'⟩)⟩,
end

lemma base.inter_basis_iff_compl_inter_basis_dual (hB : M.base B) (hX : X
   ⊆ M.E . ssE):
  M.basis (B ∩ X) X ↔ M✶.basis ((M.E \ B) ∩ (M.E \ X)) (M.E \ X) :=
begin
  refine ⟨hB.compl_inter_basis_of_inter_basis, λ h, _⟩,
  simpa using hB.compl_base_dual.compl_inter_basis_of_inter_basis h,
```

```
end

lemma dual_inj {M₁ M₂ : matroid α} (h : M₁* = M₂*) : M₁ = M₂ :=
by rw [←dual_dual M₁, h, dual_dual]

@[simp] lemma dual_inj_iff {M₁ M₂ : matroid α} : M₁* = M₂* ↔ M₁ = M₂ := ⟨
   dual_inj, congr_arg _⟩

lemma eq_dual_comm {M₁ M₂ : matroid α} : M₁ = M₂* ↔ M₂ = M₁* :=
by rw [←dual_inj_iff, dual_dual, eq_comm]

lemma coindep_iff_exists (hX : X ⊆ M.E . ssE) : M.coindep X ↔ ∃ B,
   M.base B ∧ disjoint X B :=
by rw [coindep, and_iff_right hX]

lemma coindep.exists_disjoint_base (hX : M.coindep X) : ∃ B, M.base B ∧
   disjoint X B := hX.2

end basic

section equiv -- taken from equiv.lean

variables {α β α₁ α₂ α₃ : Type*} {M : matroid α} {N : matroid β}

structure iso (M₁ : matroid α₁) (M₂ : matroid α₂) extends equiv M₁.E M₂.E :
   =
  (on_base' : ∀ (B : set M₁.E), M₁.base (coe ' B) ↔ M₂.base (coe ' (to_fun
   ' B)))

infix ` ≃i `:75 := matroid.iso

instance iso.equiv_like {α β : Type*} {M₁ : matroid α} {M₂ : matroid β} :
  equiv_like (M₁ ≃i M₂) M₁.E M₂.E :=
{ coe := λ e, e.to_equiv.to_fun,
  inv := λ e, e.to_equiv.inv_fun,
  left_inv := λ e, e.to_equiv.left_inv,
  right_inv := λ e, e.to_equiv.right_inv,
  coe_injective' := λ e e' h h', by { cases e, cases e', simpa using h }   }

def iso.symm (e : M ≃i N) : N ≃i M :=
{ to_equiv := e.symm,
  on_base' := begin
```

```
    intro B,
    rw [e.on_base'],
    congr',
    exact (e.to_equiv.image_symm_image B).symm,
  end }

def iso_of_indep (e : M.E ≃ N.E)
(hi : ∀ (I : set M.E), M.indep (coe '' I) ↔ N.indep (coe '' (e '' I))) : M ≃
   i N :=
{ to_equiv := e,
  on_base' := begin
    intro B,
    simp_rw [base_iff_maximal_indep, equiv.to_fun_as_coe],
    simp only [image_subset_iff],
    simp_rw [← hi, and.congr_right_iff],
    refine λ hI, ⟨λ h I hIN hBI, _, λ h I hI hBI, _ ⟩,
    { have hIE := hIN.subset_ground,
      rw ←@subtype.range_coe _ N.E at hIE,
      obtain ⟨I, rfl⟩ := subset_range_iff_exists_image_eq.mp hIE,
      rw [← e.image_preimage I, ← hi] at hIN,
      have := h _ hIN,
      simp only [subtype.preimage_image_coe,
   subtype.image_coe_eq_image_coe_iff] at this hBI,
      simp [this hBI] },
    specialize h (coe '' (e '' (coe ⁻¹' I))),
    simp only [subtype.preimage_image_coe, equiv.preimage_image,
   subtype.image_coe_eq_image_coe_iff,
      equiv.image_eq_iff_eq, ← hi, subtype.image_preimage_coe,
      inter_eq_self_of_subset_left hI.subset_ground] at h,
    simp only [h hI hBI, subtype.image_preimage_coe,
   inter_eq_left_iff_subset],
    exact hI.subset_ground,
  end }

@[simp] lemma coe_symm (e : M ≃i N) : (e.symm : N.E → M.E) =
   e.to_equiv.symm := rfl

def iso.cast {M N : matroid α} (h : M = N) : M ≃i N :=
{ to_equiv := equiv.cast (by rw h),
  on_base' := by { subst h, simp } }

def iso.refl (M : matroid α₁) : M ≃i M :=
```

```
⟨equiv.refl M.E, by simp⟩

def iso.trans {M₁ : matroid α₁} {M₂ : matroid α₂} {M₃ : matroid α₃}
(e₁ : M₁ ≃i M₂) (e₂ : M₂ ≃i M₃) : M₁ ≃i M₃ :=
{ to_equiv := e₁.to_equiv.trans e₂.to_equiv,
  on_base' := λ B, by {
    rw [e₁.on_base', e₂.on_base'],
    convert iff.rfl,
    rw [← image_comp],
    refl } }

def iso.image (e : M ≃i N) (B : set α) : set β := coe '' (e '' (coe ⁻¹' B))

def iso.preimage (e : M ≃i N) (B : set β) : set α := coe '' (e ⁻¹' (coe ⁻¹'
    B))

@[ssE_finish_rules] lemma iso.image_subset_ground (e : M ≃i N) (X : set α
   ) : e.image X ⊆ N.E :=
subtype.coe_image_subset _ _

@[simp] lemma iso.preimage_image (e : M ≃i N) {X : set α} (hX : X ⊆ M.E
   . ssE) :
  e.preimage (e.image X) = X :=
begin
  rw ←@subtype.range_coe _ M.E at hX,
  obtain ⟨X, rfl⟩ := subset_range_iff_exists_image_eq.mp hX,
  rw [iso.image, iso.preimage],
  simp only [subtype.preimage_image_coe,
   subtype.image_coe_eq_image_coe_iff],
  exact e.to_equiv.preimage_image X,
end

lemma iso.image_eq_preimage_symm (e : M ≃i N) {X : set α} : e.image X =
   e.symm.preimage X :=
begin
  rw [iso.preimage, coe_symm, iso.image, image_eq_image
   subtype.coe_injective,
    ←image_equiv_eq_preimage_symm], refl,
end

lemma iso.preimage_eq_image_symm (e : M ≃i N) {X : set β} : e.preimage X
   = e.symm.image X :=
```

```
begin
  rw [iso.image, coe_symm, iso.preimage, image_eq_image
   subtype.coe_injective,
    ←preimage_equiv_eq_image_symm],
  refl,
end

lemma iso.image_eq_image_inter_ground (e : M ≃i N) (X : set α) : e.image
   X = e.image (X ∩ M.E) :=
by rw [iso.image, iso.image, ←preimage_inter_range, subtype.range_coe]

@[simp] lemma iso.image_ground (e : M ≃i N) : e.image M.E = N.E :=
begin
  rw [←@subtype.range_coe _ M.E, ←@subtype.range_coe _ N.E, iso.image],
  simp only [subtype.range_coe_subtype, set_of_mem_eq,
   subtype.coe_preimage_self, image_univ],
  convert image_univ,
  { exact e.to_equiv.range_eq_univ },
  simp,
end

lemma iso.image_inter (e : M ≃i N) (X Y : set α) : e.image (X ∩ Y) =
   e.image X ∩ e.image Y :=
by rw [e.image_eq_image_inter_ground, inter_inter_distrib_right,
   iso.image,
    preimage_inter, image_inter (equiv_like.injective e), image_inter
   subtype.coe_injective,
    ← iso.image, ←iso.image, ←e.image_eq_image_inter_ground, ←
   e.image_eq_image_inter_ground ]

lemma iso.preimage_compl (e : M ≃i N) (X : set β) : e.preimage X^c = M.E \
   e.preimage X :=
by rw [iso.preimage, preimage_compl, preimage_compl, compl_eq_univ_diff,
    image_diff subtype.coe_injective, image_univ, subtype.range_coe,
   iso.preimage]

lemma iso.image_compl (e : M ≃i N) (X : set α) : e.image X^c = N.E \
   e.image X :=
by rw [iso.image_eq_preimage_symm, iso.preimage_compl, ←
   iso.image_eq_preimage_symm]

lemma iso.image_diff (e : M ≃i N) (X Y : set α) : e.image (X \ Y) =
```

```
   e.image X \ e.image Y :=
by rw [diff_eq, e.image_inter, e.image_compl, diff_eq, ←inter_assoc,
   diff_eq,
  inter_eq_self_of_subset_left (e.image_subset_ground _) ]

@[simp] lemma iso.image_empty (e : M ≃i N) : e.image ∅ = ∅ :=
by simp [iso.image]

lemma iso.image_subset_image (e : M ≃i N) {X Y : set α} (hXY : X ⊆ Y) :
   e.image X ⊆ e.image Y :=
by rw [←diff_eq_empty, ←e.image_diff, diff_eq_empty.mpr hXY,
   e.image_empty]

lemma iso.image_ground_diff (e : M ≃i N) (X : set α) : e.image (M.E \ X)
   = N.E \ e.image X :=
by rw [iso.image_diff, iso.image_ground]

def iso.dual (e : M ≃i N) : M* ≃i N* :=
{ to_equiv := e.to_equiv,
  on_base' := begin
    intro B,
    rw [dual_base_iff', dual_base_iff', ← @subtype.range_coe _ M.E, ←
   @subtype.range_coe _ N.E,
      and_iff_left (image_subset_range _ _), and_iff_left
   (image_subset_range _ _),
      ← image_univ, ←image_diff subtype.coe_injective, e.on_base', ←
   image_univ,
      ← image_diff subtype.coe_injective, equiv.to_fun_as_coe,
   image_diff (equiv.injective _),
        image_univ, equiv.range_eq_univ],
  end  }

lemma iso.on_base (e : M ≃i N) {B : set α} (hI : B ⊆ M.E) : M.base B ↔
   N.base (e.image B) :=
begin
  rw ←@subtype.range_coe _ M.E at hI,
  obtain ⟨B, rfl⟩ := subset_range_iff_exists_image_eq.mp hI,
  rw [iso.image, e.on_base', equiv.to_fun_as_coe],
  convert iff.rfl using 1,
  simp only [subtype.preimage_image_coe, eq_iff_iff],
  refl,
end
```

```
lemma iso.on_indep (e : M ≃i N) {I : set α} (hI : I ⊆ M.E) :
  M.indep I ↔ N.indep (e.image I) :=
begin
  rw [indep_iff_subset_base, indep_iff_subset_base],
  split,
  { rintro ⟨B, hB, hIB⟩,
    exact ⟨e.image B, (e.on_base hB.subset_ground).mp hB,
     e.image_subset_image hIB⟩ },
  rintro ⟨B, hB, hIB⟩,
  refine ⟨e.preimage B, _, _⟩,
  { rwa [iso.preimage_eq_image_symm, ←e.symm.on_base hB.subset_ground] },
  rw [←e.preimage_image hI, e.preimage_eq_image_symm,
    e.preimage_eq_image_symm],
  apply e.symm.image_subset_image hIB,
end

end equiv

section restriction -- taken from restriction.lean

variables {α : Type*} {I J B B' B₁ B₂ X Y R : set α} {e f : α} {M N :
   matroid α}

/-- Restrict the matroid `M` to `X : set α`.  -/
def restrict (M : matroid α) (X : set α) : matroid α :=
matroid_of_indep (X ∩ M.E) (λ I, M.indep I ∧ I ⊆ X ∩ M.E) ⟨
   M.empty_indep, empty_subset _⟩
(λ I J hJ hIJ, ⟨hJ.1.subset hIJ, hIJ.trans hJ.2⟩)
(begin
  set Y := X ∩ M.E with hY_def,
  have hY : Y ⊆ M.E := inter_subset_right _ _,
  rintro I I' ⟨hI, hIY⟩ hIn hI',
  rw ←basis_iff_mem_maximals at hIn hI',
  obtain ⟨B', hB', rfl⟩ := hI'.exists_base,
  obtain ⟨B, hB, hIB, hBIB'⟩ := hI.exists_base_subset_union_base hB',

  rw [hB'.inter_basis_iff_compl_inter_basis_dual hY, diff_inter_diff] at
   hI',

  have hss : M.E \ (B' ∪ Y) ⊆ M.E \ (B ∪ Y),
  { rw [subset_diff, and_iff_right (diff_subset _ _), ←
```

```
    subset_compl_iff_disjoint_left,
       diff_eq, compl_inter, compl_compl, ←union_assoc, union_subset_iff,
       and_iff_left (subset_union_right _ _)],
     refine hBIB'.trans (union_subset (hIY.trans _)
       (subset_union_of_subset_left (subset_union_right _ _) _)),
     apply subset_union_right },

  have hi : M*.indep (M.E \ (B ∪ Y)),
  { rw [dual_indep_iff_coindep, coindep_iff_exists],
    exact ⟨B, hB, disjoint_of_subset_right (subset_union_left _ _)
    disjoint_sdiff_left⟩ },
  have h_eq := hI'.eq_of_subset_indep hi hss
    (by {rw [diff_subset_iff, union_assoc, union_diff_self, ←
    union_assoc], simp }),

  rw [h_eq, ←diff_inter_diff, ←
    hB.inter_basis_iff_compl_inter_basis_dual hY] at hI',

  have hssu : I ⊂ (B ∩ Y) := (subset_inter hIB hIY).ssubset_of_ne
    (by { rintro rfl, exact hIn hI' }),

  obtain ⟨e, heBY, heI⟩ := exists_of_ssubset hssu,
  exact ⟨e, ⟨⟨(hBIB' heBY.1).elim (λ h', (heI h').elim) id ,heBY.2⟩,heI⟩,
    (hB.indep.inter_right Y).subset (insert_subset.mpr ⟨heBY,hssu.subset⟩),
    insert_subset.mpr ⟨heBY.2,hssu.subset.trans (inter_subset_right _ _)⟩⟩,
end)
(begin
  rintro X hX I ⟨hI, hIX⟩ hIA,
  obtain ⟨J, hJ, hIJ⟩ := hI.subset_basis_of_subset (subset_inter hIA hIX),
  refine ⟨J, ⟨⟨hJ.indep,hJ.subset.trans (inter_subset_right _ _)⟩,hIJ,
    hJ.subset.trans (inter_subset_left _ _)⟩, λ B hB hJB, _⟩,
  rw hJ.eq_of_subset_indep hB.1.1 hJB (subset_inter hB.2.2 hB.1.2),
end)
( by tauto )

@[class] structure has_restrict (α β : Type*) := (restrict : α → β → α)

infix ` ‖ ` :75 :=  has_restrict.restrict

instance : has_restrict (matroid α) (set α) := ⟨λ M E, M.restrict E⟩

@[simp] lemma restrict_indep_iff : (M ‖ R).indep I ↔ M.indep I ∧ I ⊆ R :
```

```
   =
begin
  unfold has_restrict.restrict, rw [restrict],
  simp only [subset_inter_iff, matroid_of_indep_apply,
   and.congr_right_iff, and_iff_left_iff_imp],
  refine λ hI h, hI.subset_ground,
end

lemma indep.indep_restrict_of_subset (h : M.indep I) (hIR : I ⊆ R) : (M ‖
   R).indep I :=
restrict_indep_iff.mpr ⟨h,hIR⟩

lemma restrict_ground_eq' : (M ‖ R).E = R ∩ M.E := rfl

@[simp] lemma restrict_ground_eq (hR : R ⊆ M.E . ssE) : (M ‖ R).E = R :=
by rwa [restrict_ground_eq', inter_eq_left_iff_subset]

lemma restrict_restrict (R₁ R₂ : set α) : (M ‖ R₁) ‖ R₂ = M ‖ (R₁ ∩ R₂) :=
eq_of_indep_iff_indep_forall
(by rw [restrict_ground_eq', inter_comm, restrict_ground_eq',
   restrict_ground_eq', inter_right_comm])
(λ I hI, by simp [and_assoc])

lemma restrict_restrict_of_subset {R₁ R₂ : set α} (hR : R₂ ⊆ R₁) : (M ‖
   R₁) ‖ R₂ = M ‖ R₂ :=
by rw [restrict_restrict, inter_eq_self_of_subset_right hR]

lemma restrict_eq_restrict_iff {R₁ R₂ : set α} : M ‖ R₁ = M ‖ R₂ ↔ R₁ ∩
   M.E = R₂ ∩ M.E :=
begin
  simp only [eq_iff_indep_iff_indep_forall, subset_inter_iff,
   restrict_ground_eq',
    restrict_indep_iff, and.congr_right_iff, and_imp,
   and_iff_left_iff_imp],
  intros h_eq I hIR₁ hIE hI,
  rw [iff_true_intro hIR₁, true_iff],
  exact (subset_inter hIR₁ hIE).trans (h_eq.trans_subset
   (inter_subset_left _ _)),
end

lemma restrict_inter_ground (M : matroid α) (R : set α) : M ‖ (R ∩ M.E) =
    M ‖ R :=
```

```
by rw [restrict_eq_restrict_iff, inter_assoc, inter_self]

@[simp] lemma restrict_base_iff (hX : X ⊆ M.E . ssE) : (M ‖ X).base I ↔
   M.basis I X :=
begin
  rw [base_iff_mem_maximals, basis_iff_mem_maximals],
  conv {to_lhs, congr, skip, congr, skip, congr, funext, rw
   restrict_indep_iff},
  refl,
end

@[simp] lemma basis.base_restrict (h : M.basis I X) : (M ‖ X).base I :=
restrict_base_iff.mpr h

lemma basis.basis_restrict_of_subset (hI : M.basis I X) (hXY : X ⊆ Y) (hY
   : Y ⊆ M.E . ssE) :
  (M ‖ Y).basis I X :=
by { rwa [←restrict_base_iff, restrict_restrict_of_subset hXY,
   restrict_base_iff], simpa }

lemma restrict_eq_self_iff : M ‖ X = M ↔ M.E ⊆ X :=
begin
  simp only [eq_iff_indep_iff_indep_forall, restrict_indep_iff,
   and_iff_left_iff_imp,
    restrict_ground_eq', inter_eq_right_iff_subset],
  exact λ h I hI hI', hI.trans (inter_subset_left _ _),
end

noncomputable def restrict_iso {β : Type*} {N : matroid β} (i : M ≃i N)
   (R : set α) :
  M ‖ R ≃i (N ‖ i.image R) :=
let f : (M ‖ R).E → β := λ x, i ⟨x, mem_of_mem_of_subset x.prop
   (inter_subset_right _ _)⟩,
    hf : f.injective := λ x y hxy, subtype.coe_inj.mp (by simpa using
   subtype.coe_inj.mp hxy) in
  iso_of_indep ((equiv.of_injective f hf).trans (equiv.set.of_eq
    (begin
      simp_rw [restrict_ground_eq'],
      rw [inter_eq_self_of_subset_left (iso.image_subset_ground _ _),
   iso.image,
        subset_antisymm_iff],
      simp only [image_subset_iff],
```

```
      split,
      { rintro y ⟨⟨x,hx⟩, rfl⟩,
        exact ⟨i ⟨x, (inter_subset_right _ _) hx⟩, ⟨⟨x, hx.2⟩ , hx.1, rfl⟩,
   rfl⟩ },
      rintro x (hx : coe x ∈ R),
      simp only [mem_preimage, mem_range, set_coe.exists],
      refine ⟨x, ⟨hx,x.2⟩, _⟩,
      simp [subtype.coe_inj],
    end)))
  (begin
    intros I,
    simp only [image_subset_iff, subtype.coe_mk, restrict_indep_iff,
    equiv.coe_trans,
      function.comp_app, equiv.of_injective_apply, equiv.set.of_eq_apply],
    rw [i.on_indep, and_iff_left, and_iff_left],
    { convert iff.rfl using 2,
      unfold iso.image,
      rw [subset_antisymm_iff], split,
      { rintro x ⟨y, ⟨z, hz, rfl⟩, rfl⟩,
        exact ⟨i ⟨z, (inter_subset_right _ _) z.prop⟩, ⟨_, by simpa, rfl⟩,
   rfl⟩ },
      rintro x ⟨y, ⟨⟨z,hzE⟩,⟨⟨w,hw⟩, hwI, (rfl : w = z)⟩,rfl⟩, rfl⟩,
      rw [restrict_ground_eq', mem_inter_iff] at hw,
      exact ⟨⟨i ⟨w,hzE⟩, ⟨⟨i ⟨w,hzE⟩,⟨_,by simpa using hw.1,rfl⟩,rfl⟩,by
   simp⟩⟩,
        ⟨⟨w,hw⟩,hwI,rfl⟩,rfl⟩ },
    { rintro ⟨x,hxR,hxE⟩ hxI,
      simp only [mem_preimage, subtype.coe_mk],
      exact ⟨i ⟨x, _⟩,⟨_, by simpa, rfl⟩,rfl⟩ },
    { rintro ⟨x,hxR,hxE⟩ hxI, exact hxR },
    rintro _ ⟨⟨x,hxR,hxE⟩, hI, rfl⟩,
    exact hxE,
  end)

lemma basis.transfer (hIX : M.basis I X) (hJX : M.basis J X) (hXY : X ⊆
   Y) (hJY : M.basis J Y) :
  M.basis I Y :=
begin
  rw [←restrict_base_iff],
  exact (restrict_base_iff.mpr hJY).base_of_basis_supset hJX.subset
    (hIX.basis_restrict_of_subset hXY),
end
```

```
lemma indep.exists_basis_subset_union_basis (hI : M.indep I) (hIX : I ⊆
   X) (hJ : M.basis J X) :
  ∃ I', M.basis I' X ∧ I ⊆ I' ∧ I' ⊆ I ∪ J :=
begin
  obtain ⟨I', hI', hII', hI'IJ⟩ :=
    (hI.indep_restrict_of_subset hIX).exists_base_subset_union_base
   (basis.base_restrict hJ),
  exact ⟨I', restrict_base_iff.mp hI', hII', hI'IJ⟩,
end

lemma basis.base_of_base_subset (hIX : M.basis I X) (hB : M.base B) (hBX :
    B ⊆ X) : M.base I :=
hB.base_of_basis_supset hBX hIX

lemma basis.eq_exchange_of_diff_eq_singleton (hI : M.basis I X) (hJ :
   M.basis J X)
(hIJ : I \ J = {e}) :
  ∃ f ∈ J \ I, J = (insert f I) \ {e} :=
by { rw [←restrict_base_iff] at hI hJ, exact
   hI.eq_exchange_of_diff_eq_singleton hJ hIJ }

lemma basis.encard_eq_encard_of_basis (hIX : M.basis I X) (hJX : M.basis
   J X) :
  I.encard = J.encard :=
by { rw [←restrict_base_iff] at hIX hJX, exact
   hIX.encard_eq_encard_of_base hJX }

lemma basis.card_eq_card_of_basis (hIX : M.basis I X) (hJX : M.basis J X)
   : I.ncard = J.ncard :=
by { rw [←restrict_base_iff] at hIX hJX, exact hIX.card_eq_card_of_base
   hJX }

lemma indep.augment_of_finite (hI : M.indep I) (hJ : M.indep J) (hIfin :
   I.finite)
(hIJ : I.ncard < J.ncard) :
  ∃ x ∈ J, x ∉ I ∧ M.indep (insert x I) :=
begin
  obtain ⟨K, hK, hIK⟩ :=
    (hI.indep_restrict_of_subset (subset_union_left I
   J)).exists_base_supset,
  obtain ⟨K', hK', hJK'⟩ :=
```

```
    (hJ.indep_restrict_of_subset (subset_union_right I
   J)).exists_base_supset,
  have hJfin := finite_of_ncard_pos ((nat.zero_le _).trans_lt hIJ),
  rw restrict_base_iff at hK hK',
  have hK'fin := (hIfin.union hJfin).subset hK'.subset,
  have hlt :=
    hIJ.trans_le ((ncard_le_of_subset hJK' hK'fin).trans_eq
   (hK'.card_eq_card_of_basis hK)),
  obtain ⟨e,he⟩ := exists_mem_not_mem_of_ncard_lt_ncard hlt hIfin,
  exact ⟨e, (hK.subset he.1).elim (false.elim ∘ he.2) id,
    he.2, hK.indep.subset (insert_subset.mpr ⟨he.1,hIK⟩)⟩,
end

end restriction

section closure -- taken from closure.lean

variables {α : Type*} {M : matroid α} {I J B C X Y : set α} {e f x y : α}

/-- A flat is a maximal set having a given basis  -/
def flat (M : matroid α) (F : set α) : Prop :=
(∀ {I X}, M.basis I F → M.basis I X → X ⊆ F) ∧ F ⊆ M.E

lemma ground_flat (M : matroid α) : M.flat M.E :=
⟨λ _ _ _, basis.subset_ground, subset.rfl⟩

/-- The closure of a subset of the ground set is the intersection of the
    flats containing it.
  A set `X` that doesn't satisfy `X ⊆ M.E` has the junk value `M.cl X :=
    M.cl (X ∩ M.E)`. -/
def cl (M : matroid α) (X : set α) : set α := ⋂₀ {F | M.flat F ∧ X ∩ M.E
    ⊆ F}

lemma cl_def (M : matroid α) (X : set α) : M.cl X = ⋂₀ {F | M.flat F ∧ X
    ∩ M.E ⊆ F} := rfl

lemma cl_eq_sInter_of_subset (X : set α) (hX : X ⊆ M.E . ssE) :
  M.cl X = ⋂₀ {F | M.flat F ∧ X ⊆ F} :=
by rw [cl, inter_eq_self_of_subset_left hX]

lemma cl_eq_cl_inter_ground (M : matroid α) (X : set α) : M.cl X = M.cl
    (X ∩ M.E) :=
```

```
by rw [cl_def, cl_eq_sInter_of_subset]

lemma inter_ground_subset_cl (M : matroid α) (X : set α) : X ∩ M.E ⊆
   M.cl X :=
by { rw [cl_eq_cl_inter_ground], simp [cl_def] }

@[ssE_finish_rules] lemma cl_subset_ground (M : matroid α) (X : set α) :
   M.cl X ⊆ M.E :=
begin
  apply sInter_subset_of_mem,
  simp only [mem_set_of_eq, inter_subset_right, and_true],
  apply ground_flat,
end

lemma mem_cl_iff_forall_mem_flat (X : set α) (hX : X ⊆ M.E . ssE) :
  e ∈ M.cl X ↔ ∀ F, M.flat F → X ⊆ F → e ∈ F :=
by simp_rw [cl_eq_sInter_of_subset, mem_sInter, mem_set_of_eq, and_imp]

lemma subset_cl (M : matroid α) (X : set α) (hX : X ⊆ M.E . ssE) : X ⊆
   M.cl X :=
by { rw [cl_eq_sInter_of_subset, subset_sInter_iff], simp }

lemma flat.cl {F : set α} (hF : M.flat F) : M.cl F = F :=
(sInter_subset_of_mem (by simpa)).antisymm (M.subset_cl F hF.2)

@[simp] lemma cl_ground (M : matroid α) : M.cl M.E = M.E :=
(cl_subset_ground M M.E).antisymm (M.subset_cl _)

@[simp] lemma cl_cl (M : matroid α) (X : set α) : M.cl (M.cl X) = M.cl X :
   =
begin
  nth_rewrite 2 cl_eq_cl_inter_ground,
  nth_rewrite 1 cl_eq_cl_inter_ground,
  refine (M.subset_cl _ (cl_subset_ground _ _)).antisymm' (λ e he, _),
  rw mem_cl_iff_forall_mem_flat at *,
  refine λ F hF hXF, he _ hF (λ f hf, _),
  rw mem_cl_iff_forall_mem_flat at hf,
  exact hf _ hF hXF,
end

lemma cl_subset (M : matroid α) (h : X ⊆ Y) : M.cl X ⊆ M.cl Y :=
begin
```

```
  rw [cl_eq_cl_inter_ground, M.cl_eq_cl_inter_ground Y],
  refine sInter_subset_sInter _,
  simp only [ground_inter_left, set_of_subset_set_of, and_imp],
  exact λ F hF hiF, ⟨hF, subset_trans (inter_subset_inter_left _ h) hiF⟩,
end

lemma cl_mono (M : matroid α) : monotone M.cl :=
begin
  intros X Y h,
  nth_rewrite 1 cl_eq_cl_inter_ground,
  rw cl_eq_cl_inter_ground,
  apply cl_subset,
  exact inter_subset_inter_left M.E h
end

lemma cl_subset_cl (hXY : X ⊆ M.cl Y) : M.cl X ⊆ M.cl Y :=
by simpa only [cl_cl] using M.cl_subset hXY

lemma cl_subset_cl_iff_subset_cl' : (X ⊆ M.E ∧ M.cl X ⊆ M.cl Y) ↔ X ⊆
   M.cl Y :=
⟨λ h, (M.subset_cl _ h.1).trans h.2, λ h, ⟨h.trans (cl_subset_ground _
   _), cl_subset_cl h⟩ ⟩

lemma cl_subset_cl_iff_subset_cl (hX : X ⊆ M.E . ssE) : M.cl X ⊆ M.cl Y
   ↔ X ⊆ M.cl Y :=
begin
  nth_rewrite 1 [←cl_subset_cl_iff_subset_cl'],
  rw [and_iff_right hX],
end

lemma mem_cl_of_mem (M : matroid α) (h : x ∈ X) (hX : X ⊆ M.E . ssE) : x
   ∈ M.cl X :=
(M.subset_cl X hX) h

lemma mem_cl_of_mem' (M : matroid α) (h : e ∈ X) (hX : e ∈ M.E . ssE) : e
   ∈ M.cl X :=
by { rw [cl_eq_cl_inter_ground], apply mem_cl_of_mem, exact ⟨h, hX⟩ }

@[simp] lemma cl_union_cl_right_eq (M : matroid α) (X Y : set α) :
  M.cl (X ∪ M.cl Y) = M.cl (X ∪ Y) :=
begin
  refine subset_antisymm _ _,
```

```
  { rw [cl_eq_cl_inter_ground, inter_distrib_right, ←cl_cl _ (X ∪ Y) ],
    refine M.cl_subset
      (union_subset ((inter_ground_subset_cl _ _).trans (cl_subset _
   (subset_union_left _ _))) _),
      rw [ground_inter_left],
      exact (cl_subset _ (subset_union_right _ _)) },
  rw [cl_eq_cl_inter_ground, inter_distrib_right],
  exact cl_subset _ (union_subset ((inter_subset_left _ _).trans
   (subset_union_left _ _))
    ((inter_ground_subset_cl _ _).trans (subset_union_right _ _))),
end

@[simp] lemma cl_insert_cl_eq_cl_insert (M : matroid α) (e : α) (X : set
    α) :
  M.cl (insert e (M.cl X)) = M.cl (insert e X) :=
by simp_rw [←singleton_union, cl_union_cl_right_eq]

lemma indep.cl_eq_set_of_basis (hI : M.indep I) : M.cl I = {x | M.basis I
    (insert x I)} :=
begin
  set F := {x | M.basis I (insert x I)} with hF,

  have hIF : M.basis I F,
  { rw basis_iff,
    refine ⟨hI, (λ e he, by { rw [hF, mem_set_of, insert_eq_of_mem he],
   exact hI.basis_self }),
      λ J hJ hIJ hJF, hIJ.antisymm (λ e he, _)⟩,
    rw basis.eq_of_subset_indep (hJF he) (hJ.subset (insert_subset.mpr ⟨
   he, hIJ⟩))
      (subset_insert _ _) subset.rfl,
    exact mem_insert _ _,
    rw hF, rintros e ⟨_, he⟩,
    rw ←singleton_union at he,
    exact singleton_subset_iff.mp (union_subset_iff.mp he).1 },

  have hF : M.flat F,
  {
    refine ⟨
      λ J Y hJF hJY y hy, (indep.basis_of_forall_insert hI (subset_insert
   _ _) (λ e he, _) (insert_subset.mpr ⟨hJY.subset_ground hy, by ssE⟩)),
      hIF.subset_ground
    ⟩,
```

```
    refine (basis.insert_dep (hIF.transfer hJF (subset_union_right _ _)
    (hJY.basis_union hJF)) (mem_of_mem_of_subset he _)).1,
    rw [diff_subset_iff, union_diff_self, insert_subset],
    simp only [mem_union, subset_union_left, and_true],
    right, left, exact hy
  },

  rw [subset_antisymm_iff, cl, subset_sInter_iff],
  refine ⟨sInter_subset_of_mem ⟨hF, (inter_subset_left I M.E).trans
   hIF.subset⟩, _⟩,
  rintro F' ⟨hF', hIF'⟩ e (he : M.basis I (insert e I)),
  rw (inter_eq_left_iff_subset.mpr (hIF.subset.trans hIF.subset_ground))
   at hIF',
  obtain ⟨J, hJ, hIJ⟩ := hI.subset_basis_of_subset hIF' hF'.2,

  exact (hF'.1 hJ (he.basis_union_of_subset hJ.indep hIJ)) (or.inr
   (mem_insert _ _)),
end

lemma indep.mem_cl_iff' (hI : M.indep I) :
  x ∈ M.cl I ↔ (x ∈ M.E ∧ (M.indep (insert x I) → x ∈ I)) :=
begin
  simp_rw [hI.cl_eq_set_of_basis, mem_set_of_eq],
  refine ⟨λ h, ⟨h.subset_ground (mem_insert _ _), λ h',
   h.mem_of_insert_indep (mem_insert _ _) h'⟩,
    λ h, _⟩,
  refine hI.basis_of_forall_insert (subset_insert x I) (λ e he hei, he.2
   _)
    (insert_subset.mpr ⟨h.1, hI.subset_ground⟩),
  rw [←singleton_union, union_diff_right, mem_diff, mem_singleton_iff]
   at he,
  rw [he.1] at ⊢ hei,
  exact h.2 hei,
end

lemma indep.mem_cl_iff (hI : M.indep I) (hx : x ∈ M.E . ssE) :
  x ∈ M.cl I ↔ (M.indep (insert x I) → x ∈ I) :=
begin
  simp_rw [hI.mem_cl_iff', and_iff_right_iff_imp],
  intro _, exact hx
end
```

```
lemma indep.mem_cl_iff_insert_dep_or_mem (hI : M.indep I) :
  x ∈ M.cl I ↔ M.dep (insert x I) ∨ x ∈ I :=
begin
  rw [hI.mem_cl_iff', dep_iff, insert_subset, and_iff_left
   hI.subset_ground, imp_iff_not_or],
  refine (em' (x ∈ M.E)).elim (λ hxE, _) (by tauto),
  rw [iff_false_intro hxE, false_and, and_false, false_or, false_iff],
  exact not_mem_subset hI.subset_ground hxE
end

lemma indep.insert_dep_iff (hI : M.indep I) :
  M.dep (insert e I) ↔ e ∈ M.cl I \ I :=
begin
  rw [mem_diff, hI.mem_cl_iff_insert_dep_or_mem, or_and_distrib_right,
   and_not_self, or_false,
    iff_self_and],
  refine λ hd heI, hd.not_indep _,
  rwa [insert_eq_of_mem heI],
end

lemma indep.mem_cl_iff_of_not_mem (hI : M.indep I) (heI : e ∉ I) :
  e ∈ M.cl I ↔ M.dep (insert e I) :=
by { rw [hI.mem_cl_iff', dep_iff, insert_subset, and_iff_left
   hI.subset_ground], tauto }

lemma indep.not_mem_cl_iff (hI : M.indep I) (he : e ∈ M.E . ssE) :
  e ∉ M.cl I ↔ (e ∉ I ∧ M.indep (insert e I)) :=
by rw [←not_iff_not, not_not_mem, and_comm, not_and, hI.mem_cl_iff,
   not_not_mem]

lemma Inter_cl_eq_cl_Inter_of_Union_indep {ι : Type*} (I : ι → set α) [hι
   : nonempty ι]
  (h : M.indep (⋃ i, I i)) : (∩ i, M.cl (I i)) = M.cl (∩ i, I i) :=
begin
  have hi : ∀ i, M.indep (I i), from λ i, h.subset (subset_Union _ _),
  refine subset.antisymm _ (subset_Inter (λ i, M.cl_subset (Inter_subset
   _ _))),
  rintro e he, rw mem_Inter at he,
  by_contra h',
  obtain i₀ := hι.some,
  have hiu : (∩ i , I i) ⊆ ⋃ i, I i, from
    ((Inter_subset _ i₀).trans (subset_Union _ i₀)),
```

```
have hi_inter : M.indep (∩ i, I i), from (hi i₀).subset (Inter_subset _
 _),
rw hi_inter.not_mem_cl_iff ((M.cl_subset_ground (I i₀)) (he i₀)) at h',
rw mem_Inter at h',
rw not_forall at h',
{ obtain ⟨⟨i₁, hei₁⟩, hei⟩ := h',

  have hdi₁ : ¬M.indep (insert e (I i₁)),
    from λ h_ind, hei₁ (((hi i₁).mem_cl_iff'.mp (he i₁)).2 h_ind),

  have heu : e ∉ ⋃ i, I i, from λ he, hdi₁ (h.subset
 (insert_subset.mpr ⟨he, subset_Union _ _⟩)),

  have hd_all : ∀ i, ¬M.indep (insert e (I i)),
    from λ i hind, heu (mem_Union_of_mem _ (((hi i).mem_cl_iff'.mp (he
 i)).2 hind)),

  have hb : M.basis (⋃ i, I i) (insert e (⋃ i, I i)),
  { have h' := (M.cl_subset) (subset_Union _ _) (he i₀),
    rwa [h.cl_eq_set_of_basis] at h' },

  obtain ⟨I', hI', hssI', hI'ss⟩ :=
    hei.exists_basis_subset_union_basis (insert_subset_insert hiu) hb,

  rw [insert_union, union_eq_right_iff_subset.mpr hiu] at hI'ss,

  have hI'I : I' \ (⋃ i, I i) = {e},
  { refine subset.antisymm _ (singleton_subset_iff.mpr ⟨hssI'
 (mem_insert _ _),heu⟩),
    rwa [diff_subset_iff, union_singleton] },

  obtain ⟨f, hfI, hf⟩ := hI'.eq_exchange_of_diff_eq_singleton hb hI'I,

  have hf' : ∀ i, f ∈ I i,
  { refine λ i, by_contra (λ hfi, (hd_all i (hI'.indep.subset
 (insert_subset.mpr ⟨_,_⟩)))),
    { exact hssI' (mem_insert _ _) },
    rw [←diff_singleton_eq_self hfi, diff_subset_iff, singleton_union],
    exact ((subset_Union _ i).trans_eq hf).trans (diff_subset _ _) },

  exact hfI.2 (hssI' (or.inr (by rwa mem_Inter))),
},
```

```
end

lemma bInter_cl_eq_cl_sInter_of_sUnion_indep (Is : set (set α)) (hIs :
   Is.nonempty) (h : M.indep (⋃₀ Is)) :
  (∩ I ∈ Is, M.cl I) = M.cl (∩₀ Is) :=
begin
  rw [sUnion_eq_Union] at h,
  rw [bInter_eq_Inter, sInter_eq_Inter],
  haveI := hIs.to_subtype,
  exact Inter_cl_eq_cl_Inter_of_Union_indep (λ (x : Is), coe x) h,
end

lemma basis.cl (hIX : M.basis I X) : M.cl I = M.cl X :=
(M.cl_subset hIX.subset).antisymm (cl_subset_cl
  (λ x hx, hIX.indep.mem_cl_iff.mpr (λ h, hIX.mem_of_insert_indep hx h)))

lemma basis.mem_cl_iff (hIX : M.basis I X) (he : e ∈ M.E . ssE) :
  e ∈ M.cl X ↔ (M.indep (insert e I) → e ∈ I) :=
by rw [←hIX.cl, hIX.indep.mem_cl_iff]

lemma basis.subset_cl (hI : M.basis I X) : X ⊆ M.cl I :=
by { rw hI.cl, exact M.subset_cl X hI.subset_ground }

lemma indep.basis_cl (hI : M.indep I) : M.basis I (M.cl I) :=
begin
  refine hI.basis_of_forall_insert (M.subset_cl I hI.subset_ground) (λ e
   he heI, he.2 _),
  rw [mem_diff, hI.mem_cl_iff] at he,
  obtain ⟨he, he'⟩ := he,
  rw hI.mem_cl_iff_of_not_mem he' at he,
  exact (he.not_indep heI).elim,
end

lemma indep.basis_of_subset_cl (hI : M.indep I) (hIX : I ⊆ X) (h : X ⊆
   M.cl I) : M.basis I X :=
hI.basis_cl.basis_subset hIX h

lemma indep.base_of_cl_eq_ground (hI : M.indep I) (h : M.cl I = M.E) :
   M.base I :=
by { rw base_iff_basis_ground, exact hI.basis_of_subset_cl
   hI.subset_ground (by rw h) }
```

```
lemma base.cl (hB : M.base B) : M.cl B = M.E :=
by { rw [(base_iff_basis_ground.mp hB).cl], exact M.cl_ground }

lemma base.mem_cl (hB : M.base B) (e : α) (he : e ∈ M.E . ssE) : e ∈
   M.cl B :=
by rwa [base.cl hB]

lemma cl_diff_singleton_eq_cl (h : e ∈ M.cl (X \ {e})) : M.cl (X \ {e}) =
   M.cl X :=
begin
  rw [cl_eq_cl_inter_ground, diff_eq, inter_right_comm, ←diff_eq] at *,
  rw [M.cl_eq_cl_inter_ground X],
  set X' := X ∩ M.E with hX',
  refine (M.cl_mono (diff_subset _ _)).antisymm _,

  have : (X' \ { e } ⊆ M.cl (X' \ { e })),
    { refine M.subset_cl (X' \ { e }) _,
      have g := inter_subset_right X M.E,
      have : X' \ {e} ⊆ X' := by simp only [diff_singleton_subset_iff,
   subset_insert],
      rw ←hX' at g, exact this.trans g, },
  have h' := M.cl_mono (insert_subset.mpr ⟨h, this⟩),
  rw [insert_diff_singleton, cl_cl] at h',
  exact (M.cl_mono (subset_insert _ _)).trans h',
end

lemma mem_cl_diff_singleton_iff_cl (he : e ∈ X) (heE : e ∈ M.E . ssE):
  e ∈ M.cl (X \ {e}) ↔ M.cl (X \ {e}) = M.cl X :=
begin
  rw [cl_eq_cl_inter_ground, M.cl_eq_cl_inter_ground X, diff_eq,
   inter_right_comm, ←diff_eq],
  refine ⟨cl_diff_singleton_eq_cl, λ h, _⟩,
  rw [h],
  exact M.mem_cl_of_mem' ⟨he, heE⟩,
end

lemma indep_iff_cl_diff_ne_forall : M.indep I ↔ (∀ e ∈ I, M.cl (I \ {e})
   ≠ M.cl I) :=
begin
  refine ⟨λ hI e heI h_eq, _, λ h, _⟩,
  { have hecl : e ∈ M.cl I := M.subset_cl I (hI.subset_ground) heI,
    rw [←h_eq] at hecl,
```

```
    simpa only [(hI.diff {e}).mem_cl_iff, insert_diff_singleton,
   not_mem_diff_singleton,
      insert_eq_of_mem heI, imp_iff_right hI] using hecl },
  have hIE : I ⊆ M.E,
  { refine λ e he, by_contra (λ heE, h e he _),
    rw [M.cl_eq_cl_inter_ground I, M.cl_eq_cl_inter_ground, diff_eq,
   inter_right_comm,
      inter_assoc, ←diff_eq, diff_singleton_eq_self heE] },
  obtain ⟨J, hJ⟩ := M.exists_basis I,
  convert hJ.indep,
  refine hJ.subset.antisymm' (λ e he, by_contra (λ heJ, _)),
  have hJIe : J ⊆ I \ {e}, from subset_diff_singleton hJ.subset heJ,
  have hcl := h e he,
  rw [ne.def, ←mem_cl_diff_singleton_iff_cl he] at hcl,
  have hcl' := not_mem_subset (M.cl_mono hJIe) hcl,
  rw [hJ.cl] at hcl',
  refine hcl' (M.subset_cl _ hJ.subset_ground he),
end

lemma indep_iff_not_mem_cl_diff_forall (hI : I ⊆ M.E . ssE) :
  M.indep I ↔ ∀ e ∈ I, e ∉ M.cl (I \ {e}) :=
begin
  rw indep_iff_cl_diff_ne_forall,
  { refine ⟨λ h x hxI, by { rw mem_cl_diff_singleton_iff_cl hxI, exact h
    x hxI },
            λ h x hxI, by { rw [ne.def, ←mem_cl_diff_singleton_iff_cl
    hxI], exact h x hxI }⟩, },
end

lemma indep.insert_indep_iff_of_not_mem (hI : M.indep I) (he : e ∉ I)
    (he' : e ∈ M.E . ssE):
  M.indep (insert e I) ↔ e ∉ M.cl I :=
⟨λ h, (hI.not_mem_cl_iff he').mpr ⟨he, h⟩, λ h, ((hI.not_mem_cl_iff he').mp
    h).2⟩

lemma basis_iff_cl : M.basis I X ↔ I ⊆ X ∧ X ⊆ M.cl I ∧ ∀ J ⊆ I, X ⊆
    M.cl J → J = I :=
begin
  split,
  { refine λ h, ⟨h.subset, h.subset_cl, λ J hJI hXJ, hJI.antisymm (λ e
    heI, _)⟩,
    rw [(h.indep.subset hJI).cl_eq_set_of_basis] at hXJ,
```

```
    exact (h.subset.trans hXJ heI : M.basis _ _).mem_of_insert_indep
    (mem_insert _ _)
      (h.indep.subset (insert_subset.mpr ⟨heI, hJI⟩)) },
  rintro ⟨hIX, hXI, hmin⟩,
  refine indep.basis_of_forall_insert _ hIX _ _,

  { rw indep_iff_cl_diff_ne_forall,
    intros e he hecl,
    rw ← hmin _ (diff_subset _ _) (hXI.trans_eq hecl.symm) at he,
    exact he.2 (mem_singleton e) },

  exact λ e he hi, he.2
    (((hi.subset (subset_insert _ _)).basis_cl).mem_of_insert_indep (hXI
    (he.1)) hi),
  exact hXI.trans (M.cl_subset_ground I),
end

lemma basis_union_iff_indep_cl : M.basis I (I ∪ X) ↔ M.indep I ∧ X ⊆
   M.cl I :=
begin
  refine ⟨λ h, ⟨h.indep, (subset_union_right _ _).trans h.subset_cl⟩, _⟩,
  rw basis_iff_cl,
  rintros ⟨hI, hXI⟩,
  refine ⟨subset_union_left _ _, union_subset (M.subset_cl I
   hI.subset_ground) hXI,
    λ J hJI hJ, by_contra (λ h', _)⟩,
  obtain ⟨e,heI,heJ⟩ := exists_of_ssubset (hJI.ssubset_of_ne h'),
  have heJ' : e ∈ M.cl J,
    from hJ (or.inl heI),
  refine indep_iff_not_mem_cl_diff_forall.mp hI e heI
   (mem_of_mem_of_subset heJ' _),
  exact M.cl_subset (subset_diff_singleton hJI heJ),
end

lemma basis_iff_indep_cl : M.basis I X ↔ M.indep I ∧ X ⊆ M.cl I ∧ I ⊆ X
   :=
⟨λ h, ⟨h.indep, h.subset_cl, h.subset⟩,
  λ h, (basis_union_iff_indep_cl.mpr ⟨h.1,h.2.1⟩).basis_subset h.2.2
   (subset_union_right _ _)⟩

variables {S T : set α}
```

```
/-- A set is `spanning` in `M` if its closure is equal to `M.E`, or
    equivalently if it contains
  a base of `M`. -/
def spanning (M : matroid α) (S : set α) := M.cl S = M.E ∧ S ⊆ M.E

@[ssE_finish_rules] lemma spanning.subset_ground (hS : M.spanning S) : S ⊆
    M.E :=
hS.2

lemma spanning.cl (hS : M.spanning S) : M.cl S = M.E :=
hS.1

lemma spanning_iff_cl (hS : S ⊆ M.E . ssE) : M.spanning S ↔ M.cl S = M.E
    :=
⟨and.left, λ h, ⟨h,hS⟩⟩

lemma not_spanning_iff_cl (hS : S ⊆ M.E . ssE) : ¬ M.spanning S ↔ M.cl
    S ⊂ M.E :=
begin
  rw [spanning_iff_cl, ssubset_iff_subset_ne, ne.def, iff_and_self,
    iff_true_intro (M.cl_subset_ground _)],
  refine λ _, trivial,
end

lemma ground_spanning (M : matroid α) : M.spanning M.E :=
⟨M.cl_ground, rfl.subset⟩

lemma spanning_iff_supset_base' : M.spanning S ↔ (∃ B, M.base B ∧ B ⊆ S)
    ∧ S ⊆ M.E :=
begin
  rw [spanning, and.congr_left_iff],
  refine λ hSE, ⟨λ h, _, _⟩,
  { obtain ⟨B, hB⟩ := M.exists_basis S,
    refine ⟨B, hB.indep.base_of_cl_eq_ground _, hB.subset⟩,
    rwa [hB.cl] },
  rintro ⟨B, hB, hBS⟩,
  rw [subset_antisymm_iff, and_iff_right (M.cl_subset_ground _), ←hB.cl],
  exact M.cl_subset hBS,
end

lemma spanning_iff_supset_base (hS : S ⊆ M.E . ssE) : M.spanning S ↔ ∃
    B, M.base B ∧ B ⊆ S :=
```

```
by rw [spanning_iff_supset_base', and_iff_left hS]


lemma coindep_iff_compl_spanning (hI : I ⊆ M.E . ssE) : M.coindep I ↔
   M.spanning (M.E \ I) :=
begin
  simp_rw [coindep_iff_exists, spanning_iff_supset_base, subset_diff,
   disjoint.comm],
  exact ⟨Exists.imp (λ B hB, ⟨hB.1, hB.1.subset_ground, hB.2⟩),
    Exists.imp (λ B hB, ⟨hB.1, hB.2.2⟩)⟩,
end

lemma coindep_iff_cl_compl_eq_ground (hK : X ⊆ M.E . ssE) : M.coindep X ↔
    M.cl (M.E \ X) = M.E :=
by rw [coindep_iff_compl_spanning, spanning_iff_cl]

lemma coindep.cl_compl (hX : M.coindep X) : M.cl (M.E \ X) = M.E :=
(coindep_iff_cl_compl_eq_ground hX.subset_ground).mp hX

end closure


section circuit -- taken from circuit.lean

variables {α : Type*} {M M₁ M₂ : matroid α}
  {I C C' C₁ C₂ X : set α} {e f : α}

lemma circuit.dep (hC : M.circuit C) : M.dep C := hC.1

@[ssE_finish_rules] lemma circuit.subset_ground (hC : M.circuit C) : C ⊆
   M.E :=
hC.dep.subset_ground

lemma circuit.ssubset_indep (hC : M.circuit C) (hXC : X ⊂ C) : M.indep X :
   =
begin
  rw [← not_dep_iff (hXC.subset.trans hC.subset_ground)],
  rw [circuit, mem_minimals_set_of_iff] at hC,
  exact λ h, hXC.ne.symm (hC.2 h hXC.subset),
end

lemma circuit_iff : M.circuit C ↔ M.dep C ∧ (∀ I, M.dep I → I ⊆ C → I
```

```
   = C) :=
by { rw [circuit, mem_minimals_set_of_iff], tauto }

lemma circuit_iff_forall_ssubset : M.circuit C ↔ M.dep C ∧ ∀ I ⊂ C,
   M.indep I :=
begin
  simp_rw [circuit_iff, ssubset_iff_subset_ne, and.congr_right_iff],
  exact λ hC, ⟨λ h I hIC, indep_of_not_dep (hIC.2 ∘ (λ hD, h _ hD hIC.1))
    (hIC.1.trans hC.subset_ground),
    λ h I hD hIC, by_contra (λ hne, hD.not_indep (h _ ⟨hIC, hne⟩))⟩,
end

lemma circuit.diff_singleton_indep (hC : M.circuit C) (he : e ∈ C) :
   M.indep (C \ {e}) :=
hC.ssubset_indep (diff_singleton_ssubset.2 he)

lemma circuit.diff_singleton_basis (hC : M.circuit C) (he : e ∈ C) :
   M.basis (C \ {e}) C :=
begin
  refine (hC.diff_singleton_indep he).basis_of_forall_insert (diff_subset
   _ _) (λ f hf hI, _),
  simp only [mem_diff, mem_singleton_iff, not_and, not_not] at hf,
  have := hf.2 (hf.1), subst this,
  rw [insert_diff_singleton, insert_eq_of_mem he] at hI,
  exact hC.dep.not_indep hI,
end

lemma circuit.nonempty (hC : M.circuit C) : C.nonempty :=
by {rw set.nonempty_iff_ne_empty, rintro rfl, exact hC.1.1 M.empty_indep}

lemma empty_not_circuit (M : matroid α) : ¬M.circuit ∅ :=
λ h, by simpa using h.nonempty

lemma circuit_iff_dep_forall_diff_singleton_indep :
  M.circuit C ↔ M.dep C ∧ ∀ e ∈ C, M.indep (C \ {e}) :=
begin
  rw [circuit_iff_forall_ssubset, and.congr_right_iff],
  refine λ hdep, ⟨λ h e heC, (h _ $ diff_singleton_ssubset.2 heC), λ h I
   hIC, _⟩,
  obtain ⟨e, heC,heI⟩ := exists_of_ssubset hIC,
  exact (h e heC).subset (subset_diff_singleton hIC.subset heI),
end
```

```
lemma circuit.eq_of_dep_subset_self (hC : M.circuit C) (hX : M.dep X)
    (hXC : X ⊆ C) : C = X :=
by_contra (λ h, hX.not_indep (hC.ssubset_indep (ssubset_of_subset_of_ne
    hXC (ne.symm h))))

lemma circuit.eq_of_subset_circuit (hC₁ : M.circuit C₁) (hC₂ : M.circuit
    C₂) (h : C₁ ⊆ C₂) :
  C₁ = C₂ :=
(hC₂.eq_of_dep_subset_self hC₁.dep h).symm

/-- For an independent set `I` that spans a point `e ∉ I`, the unique
    circuit contained in
`I ∪ {e}`. Has the junk value `{e}` if `e ∈ I` and `univ` if `e ∉ M.cl I`. -/
def fund_circuit (M : matroid α) (e : α) (I : set α) := insert e (⋂₀ {J |
    J ⊆ I ∧ e ∈ M.cl J})

lemma fund_circuit_subset_ground (heI : e ∈ M.cl I) : M.fund_circuit e I
    ⊆ M.E :=
begin
  refine (insert_subset.mpr ⟨(cl_subset_ground _ _) heI,
    (sInter_subset_of_mem _).trans (inter_subset_right I M.E)⟩),
  refine ⟨inter_subset_left _ _,_⟩,
  rwa [←cl_eq_cl_inter_ground],
end

lemma fund_circuit_subset_insert (he : e ∈ M.cl I) :
  M.fund_circuit e I ⊆ insert e I :=
insert_subset_insert (sInter_subset_of_mem ⟨rfl.subset, he⟩)

lemma mem_fund_circuit (M : matroid α) (e : α) (I : set α) : e ∈
    fund_circuit M e I :=
  mem_insert _ _

/-- The fundamental circuit of `e` and `I` has the junk value `{e}` if `e ∈
    I` -/
lemma indep.fund_circuit_eq_of_mem (hI : M.indep I) (he : e ∈ I) :
    M.fund_circuit e I = {e} :=
begin
  rw [fund_circuit, ←union_singleton, union_eq_right_iff_subset],
  refine sInter_subset_of_mem _,
  simp only [mem_set_of_eq, singleton_subset_iff, and_iff_right he],
```

```
  exact mem_cl_of_mem _ (mem_singleton _) (singleton_subset_iff.mpr (by
   ssE)),
end

lemma indep.fund_circuit_circuit (hI : M.indep I) (he : e ∈ M.cl I \ I) :
  M.circuit (M.fund_circuit e I) :=
begin
  rw [circuit_iff_dep_forall_diff_singleton_indep,
    ←not_indep_iff (fund_circuit_subset_ground he.1), fund_circuit],
  have hu : M.indep (⋃₀ {J : set α | J ⊆ I ∧ e ∈ M.cl J}),
    from hI.subset (sUnion_subset (λ J, and.left)),
  have hI' : I ∈ {J : set α | J ⊆ I ∧ e ∈ M.cl J}, from ⟨rfl.subset,
   he.1⟩,
  refine ⟨λ hi, _, λ f hf, _⟩,
  { rw [indep.insert_indep_iff_of_not_mem, ←
   bInter_cl_eq_cl_sInter_of_sUnion_indep _ ⟨I, hI'⟩ hu]
      at hi,
    { simpa using hi },
    { exact hI.subset (sInter_subset_of_mem hI') },
    exact λ heIs, he.2 (sInter_subset_of_mem hI' heIs) },
  obtain (rfl | hne) := em (e = f),
  { refine hu.subset _,
    simp only [insert_diff_of_mem, mem_singleton],
    exact subset_trans (diff_subset _ _)
      ((sInter_subset_of_mem hI').trans (subset_sUnion_of_mem hI')) },
  rw [mem_insert_iff, mem_sInter, eq_comm, iff_false_intro hne, false_or]
   at hf,

  have hi : M.indep (⋂₀ {J : set α | J ⊆ I ∧ e ∈ M.cl J} \ {f}),
  { exact hI.subset ((diff_subset _ _).trans (sInter_subset_of_mem hI')) },
  rw [←insert_diff_singleton_comm hne, hi.insert_indep_iff_of_not_mem],

  { intro hcl,
    exact (hf _ ⟨(diff_subset _ _).trans (sInter_subset_of_mem hI'),
   hcl⟩).2 rfl,  },

  exact λ h'e, he.2 ((diff_subset _ _).trans (sInter_subset_of_mem hI')
   h'e),
end

lemma exists_circuit_subset_of_dep (hX : M.dep X) : ∃ C ⊆ X, M.circuit C :
   =
```

```
begin
  obtain ⟨I, hI⟩ := M.exists_basis X,
  obtain (rfl | hss) := (ssubset_or_eq_of_subset hI.subset).symm,
  { exact (hX.not_indep hI.indep).elim },
  obtain ⟨e, heX, heI⟩ := exists_of_ssubset hss,
  have he : e ∈ M.cl I \ I := ⟨hI.subset_cl heX, heI⟩,
  exact ⟨M.fund_circuit e I, (fund_circuit_subset_insert he.1).trans
    (insert_subset.mpr ⟨heX, hss.subset⟩), hI.indep.fund_circuit_circuit
   he⟩,
end

lemma dep_iff_supset_circuit (hX : X ⊆ M.E . ssE) : M.dep X ↔ ∃ C ⊆ X,
   M.circuit C  :=
⟨exists_circuit_subset_of_dep, by { rintro ⟨C, hCX, hC⟩, exact
   hC.dep.supset hCX }⟩

lemma indep_iff_forall_subset_not_circuit' : M.indep I ↔ (∀ C ⊆ I, ¬
   M.circuit C) ∧ I ⊆ M.E  :=
begin
  by_cases hI : I ⊆ M.E,
  { rw [←not_iff_not, not_indep_iff],
    simp_rw [dep_iff_supset_circuit, and_iff_left hI, not_forall,
   not_not] },
  exact iff_of_false (hI ∘ indep.subset_ground) (hI ∘ and.right),
end

lemma indep_iff_forall_subset_not_circuit (hI : I ⊆ M.E . ssE) :
  M.indep I ↔ (∀ C ⊆ I, ¬ M.circuit C) :=
by rw [indep_iff_forall_subset_not_circuit', and_iff_left hI]

lemma circuit.subset_cl_diff_singleton (hC : M.circuit C) (e : α) : C ⊆
   M.cl (C \ {e}) :=
begin
  by_cases he : e ∈ C,
  { rw [(hC.diff_singleton_basis he).cl], exact M.subset_cl _ },
  rw [diff_singleton_eq_self he], exact M.subset_cl _,
end

lemma mem_cl_iff_exists_circuit (hX : X ⊆ M.E . ssE):
  e ∈ M.cl X ↔ e ∈ X ∨ ∃ C, M.circuit C ∧ e ∈ C ∧ C ⊆ insert e X :=
begin
  refine ⟨λ h, _,_⟩,
```

```
  { by_contra' h',
    obtain ⟨I, hI⟩ := M.exists_basis X,
    have hIe : ¬ M.indep (insert e I),
    { intro hI',
      refine indep_iff_not_mem_cl_diff_forall.mp hI' e (mem_insert _ _) _,
      rwa [insert_diff_of_mem _ (mem_singleton _),
        diff_singleton_eq_self (not_mem_subset hI.subset h'.1), hI.cl]},
    have heI : e ∈ M.cl I \ I, by { rw [hI.cl], exact ⟨h, not_mem_subset
   hI.subset h'.1⟩ },
    have hC := hI.indep.fund_circuit_circuit heI,
    exact h'.2 _ hC (mem_fund_circuit _ _ _)
      ((fund_circuit_subset_insert heI.1).trans (insert_subset_insert
   hI.subset)) },
  rintro (heX | ⟨C,hC, heC, hCX⟩),
  apply mem_cl_of_mem _ heX,
  refine (M.cl_subset _) (hC.subset_cl_diff_singleton e heC),
  rwa [diff_subset_iff],
end

/-- A generalization of the strong circuit elimination axiom. For finite
    matroids, this is
  equivalent to the case where `ι` is a singleton type, which is the
    usual two-circuit version.
  The stronger version is required for axiomatizing infinite matroids
    via circuits. -/
lemma circuit.strong_multi_elimination {ι : Type*} (hC : M.circuit C) (x :
   ι → α) (Cs : ι → set α)
(hCs : ∀ i, M.circuit (Cs i)) (h_mem : ∀ i, (x i) ∈ C ∩ (Cs i))
(h_unique : ∀ i i', x i ∈ Cs i' → i = i') {z : α} (hz : z ∈ C \ ⋃ i, Cs
   i) :
  ∃ C', M.circuit C' ∧ z ∈ C' ∧ C' ⊆ (C ∪ ⋃ i, (Cs i)) \ range x :=
begin
  set Y := (C ∪ ⋃ x, Cs x) \ (insert z (range x)) with hY,
  have hYE : Y ⊆ M.E,
  { refine (diff_subset _ _).trans (union_subset hC.subset_ground _),
    exact (Union_subset (λ i, (hCs i).subset_ground )) },

  have h₁ : range x ⊆ M.cl (⋃ i, ((Cs i) \ {x i} \ (insert z (range
   x)))),
  { rintro e ⟨i, rfl⟩,
    have h' := (hCs i).subset_cl_diff_singleton (x i) (h_mem i).2,
    refine mem_of_mem_of_subset h' (M.cl_subset _),
```

```
    refine subset_Union_of_subset i (subset_diff.mpr ⟨rfl.subset,_⟩),
    rw disjoint_iff_forall_ne,
    rintro y hy z (rfl | ⟨j, rfl⟩) rfl,
    { exact hz.2 (mem_Union_of_mem i hy.1) },
    refine hy.2 (mem_singleton_iff.mpr _),
    rw h_unique _ _ hy.1 },
  have h₂ : range x ⊆ M.cl Y,
  { refine h₁.trans (M.cl_subset (Union_subset (λ x, _))),
    refine diff_subset_diff_left (subset_union_of_subset_right _ _),
    exact subset_Union_of_subset x (diff_subset _ _) },
  have h₃ : C \ {z} ⊆ M.cl Y,
  { suffices : C \ {z} ⊆ (C \ insert z (range x)) ∪ (range x),
    { rw [union_diff_distrib] at hY,
      convert this.trans (union_subset_union ((subset_union_left _
   _).trans_eq hY.symm) h₂),
      rw union_eq_right_iff_subset.mpr,
      exact M.subset_cl Y },
    rw [←union_singleton, ←diff_diff, diff_subset_iff, singleton_union,
   ←insert_union,
      insert_diff_singleton, ←singleton_union, union_assoc,
    diff_union_self],
    exact subset_union_of_subset_right (subset_union_left _ _) _ },
  rw [←cl_subset_cl_iff_subset_cl] at h₃,
  have h₄ := h₃ (hC.subset_cl_diff_singleton z hz.1),
  obtain (hzY | ⟨C', hC', hzC', hCzY⟩) := mem_cl_iff_exists_circuit.mp h₄,
  { exact ((hY.subset hzY).2 (mem_insert z _)).elim },

  refine ⟨C', hC', hzC', subset_diff.mpr ⟨_,_⟩⟩,
  { exact hCzY.trans (insert_subset.mpr ⟨or.inl hz.1,diff_subset _ _⟩) },
  refine disjoint_of_subset_left hCzY _,
  rw [←singleton_union, disjoint_union_left, disjoint_singleton_left],
  refine ⟨not_mem_subset _ hz.2, _⟩,
  { rintro x' ⟨i,rfl⟩, exact mem_Union_of_mem i ((h_mem i).2) },
  exact disjoint_of_subset_right (subset_insert z _) disjoint_sdiff_left,
end

/-- The strong circuit elimination axiom. For any two circuits `C₁,C₂`
    and all `e ∈ C₁ ∩ C₂` and
  `f ∈ C₁ \ C₂`, there is a circuit `C` with `f ∈ C ⊆ (C₁ ∪ C₂) \ {e}`. -/
lemma circuit.strong_elimination (hC₁ : M.circuit C₁) (hC₂ : M.circuit C₂)
    (he : e ∈ C₁ ∩ C₂)
(hf : f ∈ C₁ \ C₂) : ∃ C ⊆ (C₁ ∪ C₂) \ {e}, M.circuit C ∧ f ∈ C :=
```

```
begin
  obtain ⟨C, hC, hfC, hCss⟩  :=
    @circuit.strong_multi_elimination _ M C₁ unit hC₁ (λ _, e) (λ _, C₂)
    (by simpa)
     (by simpa) (by simp) f (by simpa),
  simp only [range_const, Union_const] at hCss,
  exact ⟨C, hCss, hC, hfC⟩,
end

/-- The circuit eliminiation axiom : for any pair of distinct circuits `
    C₁,C₂` and any `e`, some
   circuit is contained in `C₁ ∪ C₂ \ {e}`. Traditionally this is stated
    with the assumption that
  `e ∈ C₁ ∩ C₂`, but it is also true without it. -/
lemma circuit.elimination (hC₁ : M.circuit C₁) (hC₂ : M.circuit C₂) (h :
    C₁ ≠ C₂) (e : α) :
  ∃ C ⊆ (C₁ ∪ C₂) \ {e}, M.circuit C :=
begin
  by_contra' h',
  have he : e ∈ C₁ ∩ C₂,
  { by_contra he,
    refine h' C₁ (by_contra (λ h₁, _)) hC₁,
    refine h' C₂ (by_contra (λ h₂, he _)) hC₂,
    rw [subset_diff, not_and, disjoint_singleton_right, not_not_mem] at
    h₁ h₂,
    exact ⟨h₁ (subset_union_left _ _), h₂ (subset_union_right _ _)⟩ },
  have hf : (C₁ \ C₂).nonempty,
  { rw [nonempty_iff_ne_empty, ne.def, diff_eq_empty],
    refine λ hss, h _,
    exact (hC₁.eq_of_subset_circuit hC₂ hss)},
  obtain ⟨f, hf⟩ := hf,
  obtain ⟨C, hCss, hC,-⟩ := hC₁.strong_elimination hC₂ he hf,
  exact h' C hCss hC,
end

lemma circuit.eq_fund_circuit_of_subset_insert_indep (hC : M.circuit C)
    (hI : M.indep I)
(hCI : C ⊆ insert e I) :
  C = M.fund_circuit e I :=
begin
  by_cases heE : e ∈ M.E,
  { by_contra' hne,
```

```
    have he : e ∉ I, { intro heI, rw [insert_eq_of_mem heI] at hCI,
      exact hC.dep.not_indep (hI.subset hCI) },
    have heI : e ∈ M.cl I \ I,
    { rw [mem_diff, hI.mem_cl_iff_of_not_mem he, dep_iff_supset_circuit,
   and_iff_left he],
      exact ⟨C, hCI, hC⟩ },
    obtain ⟨C', hC'ss, hC'⟩ := hC.elimination (hI.fund_circuit_circuit heI)
   hne e,
    refine hC'.dep.not_indep (hI.subset (hC'ss.trans _)),
    rw [diff_subset_iff, singleton_union],
    exact union_subset hCI (fund_circuit_subset_insert heI.1) },
  refine (hC.dep.not_indep (hI.subset (λ x hxC, (hCI hxC).elim _
   id))).elim,
  rintro rfl,
  exact (heE (hC.subset_ground hxC)).elim,
end

/-- A cocircuit is the complement of a hyperplane -/
def cocircuit (M : matroid α) (K : set α) : Prop := M✶.circuit K

@[simp] lemma dual_circuit_iff_cocircuit {K : set α} : M✶.circuit K ↔
   M.cocircuit K := iff.rfl

@[ssE_finish_rules] lemma cocircuit.subset_ground (hC : M.cocircuit C) :
   C ⊆ M.E :=
by { rw ←dual_circuit_iff_cocircuit at hC, rw ←dual_ground, exact
   hC.subset_ground }

lemma coindep_iff_forall_subset_not_cocircuit' :
  M.coindep X ↔ (∀ K ⊆ X, ¬ M.cocircuit K) ∧ X ⊆ M.E  :=
by simp [←dual_indep_iff_coindep, indep_iff_forall_subset_not_circuit']

lemma coindep_iff_forall_subset_not_cocircuit (hX : X ⊆ M.E . ssE) :
  M.coindep X ↔ (∀ K ⊆ X, ¬ M.cocircuit K) :=
by rw [coindep_iff_forall_subset_not_cocircuit', and_iff_left hX]

lemma cocircuit_iff_mem_minimals {K : set α} :
  M.cocircuit K ↔ K ∈ minimals (⊆) {X | ∀ B, M.base B → (X ∩
   B).nonempty} :=
begin
  simp_rw [cocircuit, circuit, mem_minimals_set_of_iff, dep_iff,
   dual_indep_iff_coindep,
```

```
    dual_ground, and_imp, coindep, not_and, not_exists, not_and,
   not_disjoint_iff_nonempty_inter,
    inter_comm K],
  split,
  { rintro ⟨⟨h, hKE⟩,h'⟩, refine ⟨h hKE, λ X hX hXK, h' (λ _, hX)
   (hXK.trans hKE) hXK⟩ },
  rintro ⟨h1, h2⟩,
  have hKE : K ⊆ M.E,
  { rw [←inter_eq_left_iff_subset, eq_comm],
    apply h2 (λ B hB, _) (inter_subset_left _ _),
    rw [inter_assoc, inter_eq_self_of_subset_right hB.subset_ground,
   inter_comm],
    exact h1 B hB },
  exact ⟨⟨λ _, h1,hKE⟩, λ X hX hXE hXK , h2 (hX hXE) hXK ⟩,
end

lemma cocircuit_iff_mem_minimals_compl_nonspanning {K : set α} :
  M.cocircuit K ↔ K ∈ minimals (⊆) {X | ¬M.spanning (M.E \ X) } :=
begin
  convert cocircuit_iff_mem_minimals,
  ext X,
  simp_rw [spanning_iff_supset_base, not_exists, not_and, subset_diff,
   not_and,
    not_disjoint_iff_nonempty_inter, ←and_imp, and_iff_left_of_imp
   base.subset_ground,
    inter_comm X],
end

end circuit

section flat -- taken from flat'.lean

variables {α : Type*} {M : matroid α} {I B C X Y Z K F F₀ F₁ F₂ H H₁ H₂ :
   set α}
          { e f x y z : α }

lemma flat_def : M.flat F ↔ ((∀ I X, M.basis I F → M.basis I X → X ⊆
   F) ∧ F ⊆ M.E) := iff.rfl
/- added `∧ F ⊆ M.E` to RHS.
   Here it is the last clause as in the definition, but
   in closure.lean I wrote similar assumptions
   as the first clause. -/
```

```
@[ssE_finish_rules] lemma flat.subset_ground (hF : M.flat F) : F ⊆ M.E :=
hF.2

lemma flat.eq_ground_of_spanning (hF : M.flat F) (h : M.spanning F) : F =
   M.E :=
by rw [←hF.cl, h.cl]

lemma flat.spanning_iff (hF : M.flat F) : M.spanning F ↔ F = M.E :=
⟨hF.eq_ground_of_spanning, by { rintro rfl, exact M.ground_spanning }⟩

lemma flat.Inter {ι : Type*} [hι : nonempty ι] (F : ι → set α) (hF : ∀ i,
   M.flat (F i)) :
  M.flat (⋂ i, F i) :=
begin
  split,
  { refine λ I X hI hIX, subset_Inter (λ i, _),
    obtain ⟨J, hIJ, hJ⟩ := hI.indep.subset_basis_of_subset
      ((hI.subset.trans (Inter_subset _ _ ) : I ⊆ F i)),

    have hF' := hF i, rw flat_def at hF',
    refine (union_subset_iff.mp (hF'.1 _ (F i ∪ X) hIJ _)).2,
    rw [←union_eq_left_iff_subset.mpr hIJ.subset, union_assoc],
    exact hIJ.basis_union (hIX.basis_union_of_subset hIJ.indep hJ), },
  intros e he, obtain i₀ := hι.some,
  rw mem_Inter at he,
  exact (flat.subset_ground (hF i₀)) (he i₀),
end

lemma flat_of_cl (M : matroid α) (X : set α) : M.flat (M.cl X) :=
begin
  rw [M.cl_def X, sInter_eq_Inter],
  apply flat.Inter _,
  { rintro ⟨F,hF⟩, exact hF.1 },
  use [M.E, M.ground_flat, inter_subset_right _ _],
end

lemma flat_iff_cl_self : M.flat F ↔ M.cl F = F :=
begin
  refine ⟨λ h, subset_antisymm (sInter_subset_of_mem ⟨h, inter_subset_left
   F M.E⟩)
    (M.subset_cl F (flat.subset_ground h)),
```

```
    λ h, by { rw ← h, exact flat_of_cl _ _ }⟩
end

lemma flat_iff_ssubset_cl_insert_forall (hF : F ⊆ M.E . ssE) :
  M.flat F ↔ ∀ e ∈ M.E \ F, M.cl F ⊂ M.cl (insert e F) :=
begin
  refine ⟨λ h e he, (M.cl_subset (subset_insert _ _)).ssubset_of_ne _, λ
   h, _⟩,
  { rw [h.cl],
    refine λ h', mt ((set.ext_iff.mp h') e).mpr (not_mem_of_mem_diff he)
                    ((M.subset_cl _ _) (mem_insert _ _)),
    rw insert_eq,
    refine union_subset _ hF,
    rw singleton_subset_iff, exact he.1
  },
  rw flat_iff_cl_self,
  by_contra h',
  obtain ⟨e,he',heF⟩ := exists_of_ssubset (ssubset_of_ne_of_subset
   (ne.symm h') (M.subset_cl F)),
  have h'' := h e ⟨(M.cl_subset_ground F) he', heF⟩,
  rw [←(M.cl_insert_cl_eq_cl_insert e F), insert_eq_of_mem he', M.cl_cl]
   at h'',
  exact h''.ne rfl
end

lemma flat.cl_subset_of_subset (hF : M.flat F) (h : X ⊆ F) : M.cl X ⊆ F :
   =
by { have h' := M.cl_mono h, rwa hF.cl at h' }

/-- A flat is covered by another in a matroid if they are strictly
   nested, with no flat
  between them . -/
def covby (M : matroid α) (F₀ F₁ : set α) : Prop :=
  M.flat F₀ ∧ M.flat F₁ ∧ F₀ ⊂ F₁ ∧ ∀ F, M.flat F → F₀ ⊆ F → F ⊆ F₁ →
   F = F₀ ∨ F = F₁

lemma covby_iff :
  M.covby F₀ F₁ ↔ M.flat F₀ ∧ M.flat F₁ ∧ F₀ ⊂ F₁ ∧
    ∀ F, M.flat F → F₀ ⊆ F → F ⊆ F₁ → F = F₀ ∨ F = F₁ :=
iff.rfl

lemma covby.flat_left (h : M.covby F₀ F₁) : M.flat F₀ := h.1
```

```
lemma covby.flat_right (h : M.covby F₀ F₁) : M.flat F₁ := h.2.1

lemma covby.ssubset (h : M.covby F₀ F₁) : F₀ ⊂ F₁ := h.2.2.1

lemma covby.eq_of_ssubset_of_subset (h : M.covby F₀ F₁) (hF : M.flat F)
    (hF₀ : F₀ ⊂ F)
(hF₁ : F ⊆ F₁) :
  F = F₁ :=
(h.2.2.2 F hF hF₀.subset hF₁).elim (λ h', (hF₀.ne.symm h').elim) id

lemma covby.cl_insert_eq  (h : M.covby F₀ F₁) (he : e ∈ F₁ \ F₀) :
  M.cl (insert e F₀) = F₁ :=
begin
  refine h.eq_of_ssubset_of_subset (M.flat_of_cl _)
    ((ssubset_insert he.2).trans_subset (M.subset_cl _ _))
    (h.flat_right.cl_subset_of_subset (insert_subset.mpr ⟨he.1,
   h.ssubset.subset⟩)),
  rw [insert_eq, union_subset_iff, singleton_subset_iff],
  exact ⟨h.flat_right.subset_ground he.1, h.flat_left.subset_ground⟩
end

/-- A hyperplane is a maximal set containing no base  -/
def hyperplane (M : matroid α) (H : set α) : Prop := M.covby H M.E

@[ssE_finish_rules] lemma hyperplane.subset_ground (hH : M.hyperplane H) :
    H ⊆ M.E :=
hH.flat_left.subset_ground

lemma hyperplane_iff_covby : M.hyperplane H ↔ M.covby H M.E := iff.rfl

lemma hyperplane.covby (h : M.hyperplane H) : M.covby H M.E :=
h

lemma hyperplane.flat (hH : M.hyperplane H) : M.flat H :=
hH.covby.flat_left

lemma hyperplane.ssubset_ground (hH : M.hyperplane H) : H ⊂ M.E :=
hH.covby.ssubset

lemma hyperplane.cl_insert_eq (hH : M.hyperplane H) (heH : e ∉ H) (he : e
   ∈ M.E . ssE) :
```

```
  M.cl (insert e H) = M.E :=
hH.covby.cl_insert_eq ⟨he, heH⟩

lemma hyperplane.cl_eq_univ_of_ssupset (hH : M.hyperplane H) (hX : H ⊂ X)
  (hX' : X ⊆ M.E . ssE) : M.cl X = M.E :=
begin
  obtain ⟨e, heX, heH⟩ := exists_of_ssubset hX,
  exact (M.cl_subset_ground _).antisymm ((hH.cl_insert_eq heH (hX'
   heX)).symm.trans_subset
    (M.cl_subset (insert_subset.mpr ⟨heX, hX.subset⟩))),
end

lemma hyperplane.spanning_of_ssupset (hH : M.hyperplane H) (hX : H ⊂ X)
   (hXE : X ⊆ M.E . ssE ) :
  M.spanning X :=
by rw [spanning_iff_cl, hH.cl_eq_univ_of_ssupset hX]

lemma hyperplane.not_spanning (hH : M.hyperplane H) : ¬M.spanning H :=
by { rw hH.flat.spanning_iff, exact hH.ssubset_ground.ne }

lemma hyperplane_iff_maximal_nonspanning :
  M.hyperplane H ↔ H ∈ maximals (⊆) {X | X ⊆ M.E ∧ ¬ M.spanning X } :=
begin
  simp_rw [mem_maximals_set_of_iff, and_imp],
  refine ⟨λ h, ⟨⟨h.subset_ground, h.not_spanning⟩, λ X hX hX' hHX, _⟩, λ h,
   _⟩,
  { exact by_contra (λ hne, hX' (h.spanning_of_ssupset (hHX.ssubset_of_ne
   hne))) },
  rw [hyperplane_iff_covby, covby_iff, and_iff_right M.ground_flat,
    flat_iff_ssubset_cl_insert_forall h.1.1],
  refine ⟨λ e he, _, h.1.1.ssubset_of_ne (by { rintro rfl, exact h.1.2
   M.ground_spanning }),
    λ F hF hHF hFE, or_iff_not_imp_right.mpr (λ hFE', _)⟩,
  { have h' := h.2 (insert_subset.mpr ⟨he.1, h.1.1⟩),
    simp_rw [subset_insert, forall_true_left, @eq_comm _  H,
   insert_eq_self,
      iff_false_intro he.2, imp_false, not_not, spanning_iff_cl] at h',
    rw [h', ←not_spanning_iff_cl h.1.1],
    exact h.1.2 },
  have h' := h.2 hFE,
  rw [hF.spanning_iff] at h',
  rw [h' hFE' hHF],
```

```
end

@[simp] lemma compl_cocircuit_iff_hyperplane (hH : H ⊆ M.E . ssE) :
  M.cocircuit (M.E \ H) ↔ M.hyperplane H :=
begin
  simp_rw [cocircuit_iff_mem_minimals_compl_nonspanning,
   hyperplane_iff_maximal_nonspanning,
    mem_maximals_set_of_iff, mem_minimals_set_of_iff,
   sdiff_sdiff_right_self, inf_eq_inter,
    ground_inter_right, and_imp, and_iff_right hH, and.congr_right_iff,
   subset_diff],
  refine λ hH', ⟨λ h X hX hXE hXH, _, λ h X hX hXE , _⟩,
  { rw ←diff_eq_diff_iff_eq (hXH.trans hX) hX,
    exact @h (M.E \ X) (by simpa) ⟨(diff_subset _ _),
      disjoint_of_subset_right hXH disjoint_sdiff_left⟩ },
  rw [@h (M.E \ X) (diff_subset _ _) hX, sdiff_sdiff_right_self,
   inf_eq_inter,
    inter_eq_self_of_subset_right hXE.1],
  rw [subset_diff, and_iff_right hH],
  exact hXE.2.symm,
end

@[simp] lemma compl_hyperplane_iff_cocircuit (h : K ⊆ M.E . ssE ) :
  M.hyperplane (M.E \ K) ↔ M.cocircuit K :=
by rw [←compl_cocircuit_iff_hyperplane, diff_diff_right, diff_self,
   empty_union,
  inter_comm, (inter_eq_left_iff_subset.mpr h)]

end flat

section loop -- taken from loop.lean

variables {α : Type*} {M M₁ M₂ : matroid α} {I C X Y Z K F F₁ F₂ : set α}
   {e f x y z : α}

/-- A loop is a member of the closure of the empty set -/
def loop (M : matroid α) (e : α) : Prop := e ∈ M.cl ∅

lemma loop_iff_mem_cl_empty : M.loop e ↔ e ∈ M.cl ∅ := iff.rfl

@[ssE_finish_rules] lemma loop.mem_ground (he : M.loop e) : e ∈ M.E :=
   cl_subset_ground M ∅ he
```

```
lemma loop_iff_dep : M.loop e ↔ M.dep {e} :=
by rw [loop_iff_mem_cl_empty,
  M.empty_indep.mem_cl_iff_of_not_mem (not_mem_empty e), insert_emptyc_eq]

lemma loop.dep (he : M.loop e) : M.dep {e} :=
loop_iff_dep.mp he

lemma loop_iff_circuit : M.loop e ↔ M.circuit {e} :=
begin
  by_cases he : e ∈ M.E,
  { simp_rw [circuit_iff_forall_ssubset, ssubset_singleton_iff,
   forall_eq, empty_indep, and_true,
      loop_iff_dep] },
  exact iff_of_false (he ∘ loop.mem_ground) (he ∘ (λ h, h.subset_ground
   rfl)),
end

lemma loop.not_indep_of_mem (he : M.loop e) (h : e ∈ X) : ¬ M.indep X :=
λ hX, he.dep.not_indep (hX.subset (singleton_subset_iff.mpr h))

lemma loop.not_mem_of_indep (he : M.loop e) (hI : M.indep I) : e ∉ I :=
λ h, he.not_indep_of_mem h hI

lemma loop_iff_not_indep (he : e ∈ M.E . ssE) : M.loop e ↔ ¬ M.indep {e}
    :=
by rw [loop_iff_dep, ←not_indep_iff]

/- ### Nonloops -/

/-- A `nonloop` is an element that is not a loop -/
def nonloop (M : matroid α) (e : α) : Prop := ¬ M.loop e ∧ e ∈ M.E

@[ssE_finish_rules] lemma nonloop.mem_ground (h : M.nonloop e) : e ∈ M.E :
   = h.2

lemma nonloop.not_loop (he : M.nonloop e) : ¬ M.loop e :=
he.1

lemma loop.not_nonloop (he : M.loop e) : ¬ M.nonloop e :=
λ h, h.not_loop he
```

```
@[simp] lemma not_loop_iff (he : e ∈ M.E . ssE) : ¬ M.loop e ↔ M.nonloop
   e :=
(and_iff_left he).symm

@[simp] lemma not_nonloop_iff (he : e ∈ M.E . ssE) : ¬ M.nonloop e ↔
   M.loop e :=
by rw [←not_loop_iff, not_not]

@[simp] lemma indep_singleton : M.indep {e} ↔ M.nonloop e :=
begin
  rw [nonloop, loop_iff_dep, dep_iff, not_and, not_imp_not,
   singleton_subset_iff],
  exact ⟨λ h, ⟨λ _, h, singleton_subset_iff.mp h.subset_ground⟩, λ h, h.1
   h.2⟩,
end

alias indep_singleton ↔ indep.nonloop nonloop.indep

attribute [protected] indep.nonloop nonloop.indep

lemma indep.nonloop_of_mem (hI : M.indep I) (h : e ∈ I) : M.nonloop e :=
by { rw [←not_loop_iff], exact λ he, (he.not_mem_of_indep hI) h }

lemma cocircuit.nonloop_of_mem {K : set α} (hK : M.cocircuit K) (he : e ∈
   K) : M.nonloop e :=
begin
  have heE : e ∈ M.E := hK.subset_ground he,
  rw [←not_loop_iff],
  intro hel,
  rw [cocircuit_iff_mem_minimals, mem_minimals_set_of_iff] at hK,
  suffices : K = K \ {e}, from (this.subset he).2 rfl,
  apply hK.2 (λ B hB, _) (diff_subset _ _),
  rw [diff_eq, inter_right_comm, inter_assoc, ←diff_eq,
    diff_singleton_eq_self (hel.not_mem_of_indep hB.indep)],
  exact hK.1 B hB,
end

/- ### Coloops -/

/-- A coloop is a loop of the dual  -/
def coloop (M : matroid α) (e : α) : Prop := M✶.loop e
```

```
@[ssE_finish_rules] lemma coloop.mem_ground (he : M.coloop e) : e ∈ M.E :=
@loop.mem_ground α M* e he

lemma coloop_iff_mem_cl_empty : M.coloop e ↔ e ∈ M*.cl ∅ := iff.rfl

lemma coloop.dual_loop (he : M.coloop e) : M*.loop e := he

lemma loop.dual_coloop (he : M.loop e) : M*.coloop e := by rwa [coloop,
   dual_dual]

@[simp] lemma dual_loop_iff_coloop : M*.loop e ↔ M.coloop e :=
⟨λ h, by {rw ←dual_dual M, exact h.dual_coloop}, coloop.dual_loop⟩

lemma loop_iff_not_mem_base_forall (he : e ∈ M.E . ssE) : M.loop e ↔ ∀
   B, M.base B → e ∉ B :=
by simp_rw [loop_iff_dep, ←not_indep_iff, indep_iff_subset_base,
   not_exists,
  not_and, singleton_subset_iff]

lemma coloop_iff_forall_mem_base : M.coloop e ↔ ∀ {|B|}, M.base B → e ∈
   B :=
begin
  obtain (he | he) := (em (e ∈ M.E)).symm,
  { refine iff_of_false (he ∘ coloop.mem_ground) (he ∘ (λ h, _)),
    obtain ⟨B, hB⟩ := M.exists_base,
    exact hB.subset_ground (h hB) },
  rw [←dual_loop_iff_coloop, loop_iff_not_mem_base_forall],
  simp_rw [dual_base_iff'],
  refine ⟨λ h B hB, _, λ h B hB heB, (h hB.1).2 heB⟩,
  have he' := h (M.E \ B) ⟨_, diff_subset _ _⟩,
  { simp only [mem_diff, not_and, not_not_mem] at he', exact he' he },
  simp only [sdiff_sdiff_right_self, inf_eq_inter],
  rwa inter_eq_self_of_subset_right hB.subset_ground,
end

lemma coloop.mem_of_base (he : M.coloop e) {B : set α} (hB : M.base B) :
   e ∈ B :=
coloop_iff_forall_mem_base.mp he hB

lemma coloop_iff_forall_mem_cl_iff_mem (he : e ∈ M.E . ssE) :
  M.coloop e ↔ ∀ X, e ∈ M.cl X ↔ e ∈ X :=
begin
```

```
  rw coloop_iff_forall_mem_base,
  refine ⟨λ h X, _, λ h B hB, (h B).mp (by rwa hB.cl)⟩,
  rw [cl_eq_cl_inter_ground],
  refine ⟨λ hecl, _, λ heX, _⟩,
  { obtain ⟨I, hI⟩ := M.exists_basis (X ∩ M.E),
    obtain ⟨B, hB, hIB⟩ := hI.indep.exists_base_supset,
    have heB := h hB,
    rw [hI.mem_cl_iff, imp_iff_right (hB.indep.subset (insert_subset.mpr ⟨
   heB, hIB⟩))] at hecl,
    exact (hI.subset hecl).1 },
  exact mem_cl_of_mem' _ ⟨heX, he⟩,
end

lemma coloop.mem_cl_iff_mem (he : M.coloop e) : e ∈ M.cl X ↔ e ∈ X :=
coloop_iff_forall_mem_cl_iff_mem.mp he X

lemma coloop.insert_indep_of_indep (he : M.coloop e) (hI : M.indep I) :
   M.indep (insert e I) :=
(em (e ∈ I)).elim (λ h, by rwa insert_eq_of_mem h)
  (λ h, by rwa [hI.insert_indep_iff_of_not_mem h, he.mem_cl_iff_mem])

lemma union_indep_iff_indep_of_subset_coloops (hK : K ⊆ M✶.cl ∅) :
   M.indep (I ∪ K) ↔ M.indep I :=
begin
  refine ⟨λ h, h.subset (subset_union_left I K), λ h, _⟩,
  obtain ⟨B, hB, hIB⟩ := h.exists_base_supset,
  exact hB.indep.subset (union_subset hIB (hK.trans (λ e he,
   coloop.mem_of_base he hB))),
end

lemma diff_indep_iff_indep_of_subset_coloops (hK : K ⊆ M✶.cl ∅) : M.indep
   (I \ K) ↔ M.indep I :=
by rw [←union_indep_iff_indep_of_subset_coloops hK, diff_union_self,
  union_indep_iff_indep_of_subset_coloops hK]

lemma indep_iff_diff_coloops_indep : M.indep I ↔ M.indep (I \ M✶.cl ∅) :=
  (diff_indep_iff_indep_of_subset_coloops subset.rfl).symm

lemma coloops_indep (M : matroid α) : M.indep (M✶.cl ∅) :=
by { rw [indep_iff_diff_coloops_indep, diff_self], exact M.empty_indep }

lemma indep_of_subset_coloops (h : I ⊆ M✶.cl ∅) : M.indep I :=
```

```
M.coloops_indep.subset h

lemma coloop_iff_cocircuit : M.coloop e ↔ M.cocircuit {e} :=
by rw [←dual_loop_iff_coloop, loop_iff_circuit,
   dual_circuit_iff_cocircuit]

end loop

section rank -- taken from rank.lean

variables {α : Type*} {M : matroid α}  {B X Y X' Y' Z I J : set α} {e f x
   y z : α} {k n : ℕ}

/-- The rank `r X` of a set `X` is the cardinality of one of its bases, or
   zero if its bases are
  infinite -/
def er {α : Type*} (M : matroid α) (X : set α) : ℕ∞ :=
  ⨅ (I : {I | M.basis I (X ∩ M.E)}), encard (I : set α)

lemma basis.encard_of_inter_ground (hI : M.basis I (X ∩ M.E)) : I.encard =
    M.er X :=
begin
  have hrw : ∀ J : {J : set α | M.basis J (X ∩ M.E)}, (J : set α).encard
   = I.encard,
  { rintro ⟨J, hJ⟩, exact (hI.encard_eq_encard_of_basis hJ).symm },
  haveI : nonempty {J : set α | M.basis J (X ∩ M.E)},
    from let ⟨I,hI⟩ := M.exists_basis (X ∩ M.E) in ⟨⟨I,hI⟩⟩,
  simp_rw [er, hrw, cinfi_const],
end

lemma basis.encard (hI : M.basis I X) : I.encard = M.er X :=
hI.basis_inter_ground.encard_of_inter_ground

lemma eq_er_iff {n : ℕ∞} (hX : X ⊆ M.E . ssE) : M.er X = n ↔ ∃ I,
   M.basis I X ∧ I.encard = n :=
begin
  obtain ⟨I, hI⟩ := M.exists_basis X,
  rw [←hI.encard],
  refine ⟨λ h, ⟨I, hI, h⟩, _⟩,
  rintro ⟨J, hJ, rfl⟩,
  rw [hI.encard, hJ.encard],
end
```

```
lemma indep.er (hI : M.indep I) : M.er I = I.encard := eq_er_iff.mpr ⟨I,
   hI.basis_self, rfl⟩

lemma basis.er (hIX : M.basis I X) : M.er I = M.er X :=
by rw [←hIX.encard, hIX.indep.er]

lemma er_eq_er_inter_ground (M : matroid α) (X : set α) : M.er X = M.er
   (X ∩ M.E) :=
by { obtain ⟨I, hI⟩ := M.exists_basis (X ∩ M.E), rwa
   [←hI.encard_of_inter_ground, ←basis.encard] }

lemma er_mono (M : matroid α) : monotone M.er :=
begin
  rintro X Y (h : X ⊆ Y),
  rw [er_eq_er_inter_ground, M.er_eq_er_inter_ground Y],
  obtain ⟨I, hI⟩ := M.exists_basis (X ∩ M.E),
  obtain ⟨J, hJ, hIJ⟩ :=
    hI.indep.subset_basis_of_subset (hI.subset.trans
   (inter_subset_inter_left M.E h)),
  rw [←hI.encard, ←hJ.encard],
  exact encard_mono hIJ,
end

lemma indep.encard_le_er_of_subset (hI : M.indep I) (hIX : I ⊆ X) :
   I.encard ≤ M.er X :=
  by { rw [←hI.er], exact M.er_mono hIX }

lemma er_le_iff {n : ℕ∞} : M.er X ≤ n ↔ (∀ I ⊆ X, M.indep I →
   I.encard ≤ n) :=
begin
  refine ⟨λ h I hIX hI, (hI.encard_le_er_of_subset hIX).trans h, λ h, _⟩,
  obtain ⟨J, hJ⟩ := M.exists_basis (X ∩ M.E),
  rw [er_eq_er_inter_ground, ←hJ.encard],
  exact h J (hJ.subset.trans (inter_subset_left _ _)) hJ.indep,
end

@[simp] lemma er_cl (M : matroid α) (X : set α) : M.er (M.cl X) = M.er X :
   =
begin
  rw [cl_eq_cl_inter_ground, M.er_eq_er_inter_ground X],
  obtain ⟨I, hI⟩ := M.exists_basis (X ∩ M.E),
```

```
  rw [←hI.er, ←hI.cl, hI.indep.basis_cl.er],
end

lemma basis_iff_indep_encard_eq_of_finite (hIfin : I.finite) (hXE : X ⊆
   M.E . ssE) :
  M.basis I X ↔ I ⊆ X ∧ M.indep I ∧ I.encard = M.er X :=
begin
  rw [basis_iff_indep_cl, and_comm (I ⊆ X), ←and_assoc,
   and.congr_left_iff, and.congr_right_iff],
  refine λ hIX hI, ⟨λ h, (hI.encard_le_er_of_subset hIX).antisymm _, λ h,
   _⟩,
  { refine (M.er_mono h).trans _,
    rw [er_cl, hI.er], exact rfl.le },
  intros e he,
  rw [hI.mem_cl_iff (hXE he)],
  refine λ hi, by_contra (λ he', _),
  have hr := hi.er, rw [encard_insert_of_not_mem he'] at hr,
  have hle := M.er_mono (insert_subset.mpr ⟨he, hIX⟩),
  rw [hr, ←h, hIfin.encard_eq, ←nat.cast_one, ←nat.cast_add,
   nat.cast_le] at hle,
  simpa using hle,
end

def r_fin (M : matroid α) (X : set α) := M.er X < ⊤

lemma r_fin_iff_er_ne_top : M.r_fin X ↔ M.er X ≠ ⊤ :=
by rw [r_fin, ←lt_top_iff_ne_top]

lemma r_fin_iff_er_lt_top : M.r_fin X ↔ M.er X < ⊤ :=
iff.rfl

lemma r_fin_iff_inter_ground : M.r_fin X ↔ M.r_fin (X ∩ M.E) :=
by rw [r_fin, er_eq_er_inter_ground, r_fin]

lemma to_r_fin (M : matroid α) [finite_rk M] (X : set α) : M.r_fin X :=
begin
  obtain ⟨I, hI⟩ := M.exists_basis (X ∩ M.E),
  rw [r_fin_iff_inter_ground, r_fin_iff_er_lt_top, ← hI.encard,
   encard_lt_top_iff_finite],
  exact hI.finite,
end
```

```
/-- The rank function. Intended to be used in a `finite_rk` matroid;
   otherwise `er` is better.-/
def r (M : matroid α) (X : set α) : ℕ := (M.er X).to_nat

/-- The rank of the ground set of a matroid -/
@[reducible] def rk (M : matroid α) : ℕ := M.r M.E

lemma rk_def (M : matroid α) : M.rk = M.r M.E := rfl

@[simp] lemma er_to_nat_eq_r (M : matroid α) (X : set α) : (M.er
   X).to_nat = M.r X := rfl

lemma r_fin.coe_r_eq_er (hX : M.r_fin X) : (M.r X : ℕ∞) = M.er X :=
begin
  obtain ⟨I, hI⟩ := M.exists_basis (X ∩ M.E),
  rwa [r, er_eq_er_inter_ground, ←hI.encard, enat.coe_to_nat_eq_self,
   hI.encard,
    ←er_eq_er_inter_ground, ←r_fin_iff_er_ne_top],
end

@[simp] lemma coe_r_eq_er (M : matroid α) [finite_rk M] (X : set α) :
   (M.r X : ℕ∞) = M.er X :=
(M.to_r_fin X).coe_r_eq_er

@[simp] lemma er_eq_coe_iff [finite_rk M] {n : ℕ} : M.er X = n ↔ M.r X =
   n :=
by rw [←coe_r_eq_er, enat.coe_inj]

@[simp] lemma er_le_er_iff [finite_rk M] : M.er X ≤ M.er Y ↔ M.r X ≤
   M.r Y :=
by rw [←coe_r_eq_er, ←coe_r_eq_er, enat.coe_le_coe_iff]

lemma indep.r (hI : M.indep I) : M.r I = I.ncard :=
by rw [←er_to_nat_eq_r, hI.er, encard_to_nat_eq]

lemma basis_iff_indep_card [finite_rk M] (hX : X ⊆ M.E . ssE) :
  M.basis I X ↔ M.indep I ∧ I ⊆ X ∧ I.ncard = M.r X :=
begin
  refine I.finite_or_infinite.symm.elim (λ hI, iff_of_false (hI ∘ (λ h,
   h.indep.finite))
    (hI ∘ λ h, h.1.finite)) (λ hIfin, _),
  rw [basis_iff_indep_encard_eq_of_finite hIfin hX, and_comm (_ ⊆ _),
```

```
   and_assoc,
    and_comm (_ ⊆ _),  ←coe_r_eq_er, hIfin.encard_eq, enat.coe_inj],
end

lemma base_iff_indep_card [finite_rk M] : M.base B ↔ M.indep B ∧ B.ncard
   = M.rk :=
by rw [base_iff_basis_ground, basis_iff_indep_card, ←and_assoc,
  and_iff_left_of_imp indep.subset_ground]

lemma r_eq_r_inter_ground (M : matroid α) (X : set α) : M.r X = M.r (X ∩
   M.E) :=
by rw [←er_to_nat_eq_r, er_eq_er_inter_ground, er_to_nat_eq_r]

lemma r_le_iff [finite_rk M] : M.r X ≤ n ↔ (∀ I ⊆ X, M.indep I →
   I.ncard ≤ n) :=
begin
  simp_rw [←enat.coe_le_coe_iff, coe_r_eq_er, er_le_iff,
   encard_le_coe_iff, enat.coe_le_coe_iff],
  exact forall_congr (λ I, ⟨by tauto, λ h hIX hI, ⟨hI.finite, h hIX hI⟩⟩),
end

lemma r_mono (M : matroid α) [finite_rk M] : monotone M.r :=
by { rintro X Y (hXY : X ⊆ Y), rw [←er_le_er_iff],  exact M.er_mono hXY }

lemma r_le_card (M : matroid α) [finite M] (X : set α) (hX : X ⊆ M.E .
   ssE) : M.r X ≤ X.ncard :=
by { rw [r_le_iff], exact λ I h _, ncard_le_of_subset h (M.set_finite X) }

lemma r_le_rk (M : matroid α) [finite_rk M] (X : set α) : M.r X ≤ M.rk :=
by { rw [r_eq_r_inter_ground], exact M.r_mono (inter_subset_right _ _) }

lemma indep.base_of_rk_le_card [finite_rk M] (hI : M.indep I) (h : M.rk ≤
   I.ncard) : M.base I :=
base_iff_indep_card.mpr ⟨hI, h.antisymm' (by {rw ←hI.r, apply r_le_rk})⟩

lemma basis.card (h : M.basis I X) : I.ncard = M.r X :=
by rw [←encard_to_nat_eq, ←er_to_nat_eq_r, h.encard]

lemma base.card (hB : M.base B) : B.ncard = M.rk :=
by rw [(base_iff_basis_ground.mp hB).card, rk]

end rank
```

```
section simple -- taken from simple.lean

variables {α : Type*} {N M : matroid α} {e f g : α} {X Y Z S T : set α}

/-- A matroid is loopless on a set if that set doesn't contain a loop. -/
def loopless_on (M : matroid α) (X : set α) : Prop := ∀ {e}, e ∈ X →
   M.nonloop e

/-- A matroid is loopless if it has no loop -/
def loopless (M : matroid α) : Prop := ∀ e ∈ M.E, M.nonloop e

@[simp] lemma loopless_on_ground : M.loopless_on M.E ↔ M.loopless := by
   simp [loopless_on, loopless]

/-- the property of a set containing no loops or para pairs -/
def simple_on (M : matroid α) (X : set α) : Prop := ∀ {e}, e ∈ X → ∀ {
   f}, f ∈ X → M.indep {e, f}

/-- the property of a matroid having no loops or para pairs -/
def simple (M : matroid α) : Prop := ∀ (e ∈ M.E) (f ∈ M.E), M.indep {e,
   f}

@[simp] lemma simple_on_ground : M.simple_on M.E ↔ M.simple := by simp
   [simple_on, simple]

protected lemma simple_on.loopless_on (h : M.simple_on X) : M.loopless_on
   X :=
begin
  intros x hx,
  rw [←indep_singleton , ←pair_eq_singleton],
  exact h hx hx,
end

protected lemma simple.loopless (h : M.simple) : M.loopless :=
  loopless_on_ground.2 ((simple_on_ground.2 h).loopless_on)

end simple

end matroid
```

A.3 Matroid Constructions

```
    /-
Code copied from https://github.com/apnelson1/lean-matroids
-/
import .matroid_mwe

variables {α : Type*} {M : matroid α} {k a b c : ℕ} {I J X C B E : set α}

open set

namespace matroid

section delete

variables {D Y Z R : set α} {e : α} {N : matroid α}

variables {D₁ D₂ : set α}

class has_delete (α β : Type*) := (del : α → β → α)

infix ` \ ` :75 :=  has_delete.del

def delete (M : matroid α) (D : set α) : matroid α := M ‖ Dᶜ

instance del_set {α : Type*} : has_delete (matroid α) (set α) := ⟨
   matroid.delete⟩
instance del_elem {α : Type*} : has_delete (matroid α) α := ⟨λ M e,
   M.delete {e}⟩

@[simp] lemma delete_compl (M : matroid α) (R : set α) : M \ Rᶜ = M ‖ R :=
by { change M ‖ Rᶜᶜ = M ‖ R, rw compl_compl }

@[simp] lemma restrict_compl (M : matroid α) (D : set α) : M ‖ Dᶜ = M \ D
   := rfl

@[simp] lemma restrict_ground_diff (M : matroid α) (D : set α) : M ‖ (M.E
   \ D) = M \ D :=
by rw [←restrict_compl, ← M.restrict_inter_ground Dᶜ,
   diff_eq_compl_inter]

@[simp] lemma delete_ground (M : matroid α) (D : set α) : (M \ D).E = M.E
```

```
   \ D :=
by rw [←restrict_compl, restrict_ground_eq', diff_eq_compl_inter]

@[ssE_finish_rules] lemma delete_ground_subset_ground (M : matroid α) (D :
    set α) :
  (M \ D).E ⊆ M.E := (M.delete_ground D).trans_subset (diff_subset _ _)

@[simp] lemma delete_elem (M : matroid α) (e : α) : M \ e = M \ ({e} :
   set α) := rfl

@[simp] lemma delete_delete (M : matroid α) (D₁ D₂ : set α) : (M \ D₁) \
   D₂ = M \ (D₁ ∪ D₂) :=
by rw [←restrict_compl, ←restrict_compl, ←restrict_compl,
   restrict_restrict, compl_union]

lemma delete_eq_delete_iff : M \ D₁ = M \ D₂ ↔ D₁ ∩ M.E = D₂ ∩ M.E :=
by simp_rw [←restrict_compl, restrict_eq_restrict_iff,
    ←diff_eq_compl_inter, diff_eq_diff_iff_inter_eq_inter, inter_comm
   M.E]

lemma delete_eq_delete_inter_ground (M : matroid α) (D : set α) : M \ D =
   M \ (D ∩ M.E) :=
by rw [delete_eq_delete_iff, inter_assoc, inter_self]

lemma delete_eq_self_iff : M \ D = M ↔ disjoint D M.E :=
by rw [←restrict_compl, restrict_eq_self_iff,
   subset_compl_iff_disjoint_left]

@[simp] lemma delete_indep_iff : (M \ D).indep I ↔ M.indep I ∧ disjoint
   I D :=
by rw [←restrict_compl, restrict_indep_iff,
   subset_compl_iff_disjoint_right]

lemma indep.of_delete (h : (M \ D).indep I) : M.indep I :=
(delete_indep_iff.mp h).1

lemma indep.indep_delete_of_disjoint (h : M.indep I) (hID : disjoint I D)
   : (M \ D).indep I :=
delete_indep_iff.mpr ⟨h, hID⟩

@[simp] lemma delete_base_iff : (M \ D).base B ↔ M.basis B (M.E \ D) :=
by rw [←restrict_compl, ←restrict_inter_ground, ←diff_eq_compl_inter,
```

```
   restrict_base_iff]

@[simp] lemma delete_basis_iff : (M \ D).basis I X ↔ M.basis I X ∧
   disjoint X D :=
begin
  simp_rw [basis_iff', delete_indep_iff, delete_ground, subset_diff,
   and_assoc,
    and.congr_right_iff, and_imp, ←and_assoc, and.congr_left_iff],
  refine λ hI hdj hX, ⟨λ h, ⟨h.1.2, λ J hJ hIJ hJX, h.2 J hJ _ hIJ hJX⟩,
    λ h, ⟨⟨_, h.1⟩,λ J hJ hJD hIJ hJX, h.2 J hJ hIJ hJX⟩⟩,
  { exact disjoint_of_subset_left hJX hdj },
  exact disjoint_of_subset_left h.1 hdj
end

lemma basis.to_delete (h : M.basis I X) (hX : disjoint X D) : (M \
   D).basis I X :=
by { rw [delete_basis_iff], exact ⟨h, hX⟩ }

@[simp] lemma delete_dep_iff : (M \ D).dep X ↔ M.dep X ∧ disjoint X D :=
by { rw [dep_iff, dep_iff, delete_indep_iff, delete_ground, subset_diff],
   tauto! }

@[simp] lemma delete_loop_iff : (M \ D).loop e ↔ M.loop e ∧ e ∉ D :=
by rw [loop_iff_dep, delete_dep_iff, disjoint_singleton_left,
   loop_iff_dep]

@[simp] lemma delete_nonloop_iff : (M \ D).nonloop e ↔ M.nonloop e ∧ e ∉
   D :=
by rw [←indep_singleton, delete_indep_iff, disjoint_singleton_left,
   indep_singleton]

@[simp] lemma delete_circuit_iff : (M \ D).circuit C ↔ M.circuit C ∧
   disjoint C D :=
begin
  simp_rw [circuit_iff, delete_dep_iff, and_imp],
  rw [and_comm, ←and_assoc, and.congr_left_iff, and_comm,
   and.congr_right_iff],
  refine λ hdj hC, ⟨λ h I hI hIC, h I hI _ hIC, λ h I hI hdj' hIC, h I hI
   hIC⟩,
  exact disjoint_of_subset_left hIC hdj,
end
```

```
lemma fund_circuit_delete (hI : M.indep I) (heI : e ∈ M.cl I)
  (hdj : disjoint (insert e I) D) : (M \ D).fund_circuit e I =
   M.fund_circuit e I :=
begin
  by_cases e ∈ I,
  { rw [hI.fund_circuit_eq_of_mem h, (delete_indep_iff.2 ⟨hI,
      disjoint_of_subset_left (subset_insert e I)
   hdj⟩).fund_circuit_eq_of_mem h] },
  have hC : (M \ D).circuit (M.fund_circuit e I),
  { rw [delete_circuit_iff, and_iff_right (hI.fund_circuit_circuit
   ((mem_diff e).2 ⟨heI, h⟩))],
    exact disjoint_of_subset_left (fund_circuit_subset_insert ((mem_diff
   e).2 ⟨heI, h⟩).1) hdj },

  refine (hC.eq_fund_circuit_of_subset_insert_indep _
   (fund_circuit_subset_insert
    ((mem_diff e).2 ⟨heI, h⟩).1)).symm,
  rw [delete_indep_iff],
  exact ⟨hI, disjoint_of_subset_left (subset_insert _ _) hdj⟩
end

@[simp] lemma delete_cl_eq (M : matroid α) (D X : set α) : (M \ D).cl X =
   M.cl (X \ D) \ D :=
begin
  obtain ⟨I, hI⟩ := (M \ D).exists_basis ((X \ D) ∩ (M \ D).E),
  simp_rw [delete_ground, diff_eq, inter_assoc, inter_comm D^c,
   inter_assoc, inter_self,
    ←inter_assoc] at hI,
  rw [cl_eq_cl_inter_ground, delete_ground, diff_eq, ←inter_assoc, ←
   hI.cl],
  have hI' := (delete_basis_iff.mp hI).1,

  rw [M.cl_eq_cl_inter_ground, diff_eq X D, inter_right_comm, ←hI'.cl,
   set.ext_iff],
  simp_rw [hI.indep.mem_cl_iff', mem_diff, hI'.indep.mem_cl_iff',
   delete_ground, mem_diff,
    delete_indep_iff, and_assoc, and.congr_right_iff, and_comm (_ ∉ D),
   and.congr_left_iff,
    and_imp, ←union_singleton, disjoint_union_left,
   disjoint_singleton_left, union_singleton ],

  refine λ e heE heD, _,
```

```
  rw [iff_true_intro (disjoint_of_subset_left hI'.subset _),
   iff_true_intro heD],
  { simp },
  rw ←diff_eq, exact disjoint_sdiff_left,
end

lemma delete_loops_eq : (M \ D).cl ∅ = M.cl ∅ \ D :=
by simp only [delete_cl_eq, empty_diff]

lemma delete_er_eq' (M : matroid α) (D X : set α) : (M \ D).er X = M.er
   (X \ D) :=
begin
  rw [delete_eq_delete_inter_ground, er_eq_er_inter_ground,
   delete_ground, diff_inter_self_eq_diff,
    diff_eq, inter_comm M.E, ← inter_assoc, ←diff_eq,
   M.er_eq_er_inter_ground ( X \ D)],
  obtain ⟨I, hI⟩ := M.exists_basis ((X \ D) ∩ M.E),
  rw [←(hI.to_delete (disjoint_of_subset (inter_subset_left _ _)
   (inter_subset_left _ _)
    disjoint_sdiff_left)).encard, ←hI.encard],
end

@[simp] lemma delete_empty (M : matroid α) : M \ (∅ : set α) = M :=
by { rw [delete_eq_self_iff], exact empty_disjoint _ }

noncomputable def delete_iso {β : Type*} {N : matroid β} (i : M ≃i N) (D
   : set α) :
  M \ D ≃i (N \ i.image D) :=
(iso.cast (M.restrict_ground_diff D).symm).trans
  ((restrict_iso i _).trans
  (iso.cast (by rw [i.image_ground_diff, restrict_ground_diff] )))

end delete

section contract

variables {C₁ C₂ : set α}

class has_contract (α β : Type*) := (con : α → β → α)

infix ` / ` :75 :=  has_contract.con
```

```
def contract (M : matroid α) (C : set α) : matroid α := (M* \ C)*

instance con_set {α : Type*} : has_contract (matroid α) (set α) := ⟨
   matroid.contract⟩
instance con_elem {α : Type*} : has_contract (matroid α) α := ⟨λ M e,
   M.contract {e}⟩

@[simp] lemma dual_delete_dual_eq_contract (M : matroid α) (X : set α) :
   (M* \ X)* = M / X :=
rfl

@[simp] lemma contract_dual_eq_dual_delete (M : matroid α) (X : set α) :
   (M / X)* = M* \ X :=
by rw [←dual_delete_dual_eq_contract, dual_dual]

@[simp] lemma contract_ground (M : matroid α) (C : set α) : (M / C).E =
   M.E \ C :=
by rw [←dual_delete_dual_eq_contract, dual_ground, delete_ground,
   dual_ground]

@[ssE_finish_rules] lemma contract_ground_subset_ground (M : matroid α)
   (C : set α) :
  (M / C).E ⊆ M.E  := (M.contract_ground C).trans_subset (diff_subset _ _)

@[simp] lemma dual_contract_dual_eq_delete (M : matroid α) (X : set α) :
   (M* / X)* = M \ X :=
by rw [←dual_delete_dual_eq_contract, dual_dual, dual_dual]

@[simp] lemma delete_dual_eq_dual_contract (M : matroid α) (X : set α) :
   (M \ X)* = M* / X :=
by rw [←dual_delete_dual_eq_contract, dual_dual]

@[simp] lemma contract_elem (M : matroid α) (e : α) : M / e = M / ({e} :
   set α) := rfl

@[simp] lemma contract_contract (M : matroid α) (C₁ C₂ : set α) : M / C₁
   / C₂ = M / (C₁ ∪ C₂) :=
by rw [eq_comm, ←dual_delete_dual_eq_contract, ←delete_delete, ←
   dual_contract_dual_eq_delete,
    ←dual_contract_dual_eq_delete, dual_dual, dual_dual, dual_dual]

lemma indep.contract_base_iff (hI : M.indep I) : (M / I).base B ↔
```

```
   disjoint B I ∧ M.base (B ∪ I) :=
begin
  have hIE := hI.subset_ground,
  rw [←dual_dual M, dual_indep_iff_coindep, coindep_iff_exists] at hI,
  obtain ⟨B₀, hB₀, hfk⟩ := hI,
  rw [←dual_dual M, ←dual_delete_dual_eq_contract, dual_base_iff',
   dual_base_iff',
    delete_base_iff, dual_dual, delete_ground, diff_diff, union_comm,
   union_subset_iff,
     subset_diff, and_comm (disjoint _ _), ← and_assoc,
    and.congr_left_iff, dual_ground, and_iff_left hIE,
   and.congr_left_iff],

  exact λ hIB hBE, ⟨λ h, h.base_of_base_subset hB₀ (subset_diff.mpr ⟨
   hB₀.subset_ground, hfk.symm⟩),
    λ hB, hB.basis_of_subset (diff_subset _ _) (diff_subset_diff_right
   (subset_union_right _ _))⟩,
end

lemma indep.contract_indep_iff (hI : M.indep I) :
  (M / I).indep J ↔ disjoint J I ∧ M.indep (J ∪ I)  :=
begin
  simp_rw [indep_iff_subset_base, hI.contract_base_iff, union_subset_iff],
  split,
  { rintro ⟨B, ⟨hdj, hBI⟩, hJB⟩,
    exact ⟨disjoint_of_subset_left hJB hdj, _, hBI, hJB.trans
   (subset_union_left _ _),
      (subset_union_right _ _)⟩ },
  rintro ⟨hdj, B, hB, hJIB⟩,
  refine ⟨B \I, ⟨disjoint_sdiff_left, _⟩, _⟩,
  { rwa [diff_union_self, union_eq_self_of_subset_right hJIB.2] },
  rw [subset_diff],
  exact ⟨hJIB.1, hdj⟩,
end

lemma indep.union_indep_iff_contract_indep (hI : M.indep I) :
  M.indep (I ∪ J) ↔ (M / I).indep (J \ I) :=
by rw [hI.contract_indep_iff, and_iff_right disjoint_sdiff_left,
   diff_union_self, union_comm]

lemma indep.contract_dep_iff (hI : M.indep I) :
  (M / I).dep J ↔ disjoint J I ∧ M.dep (J ∪ I) :=
```

```
begin
  rw [dep_iff, hI.contract_indep_iff, dep_iff, contract_ground,
   subset_diff,
    disjoint.comm, union_subset_iff, and_iff_left hI.subset_ground],
  tauto!,
end

lemma contract_eq_delete_of_subset_coloops (hX : X ⊆ M✶.cl ∅) : M / X = M
   \ X :=
begin
  refine eq_of_indep_iff_indep_forall rfl (λ I hI, _),
  rw [(indep_of_subset_coloops hX).contract_indep_iff, delete_indep_iff,
   and_comm,
    union_indep_iff_indep_of_subset_coloops hX],
end

lemma contract_eq_self_iff : M / C = M ↔ disjoint C M.E :=
by rw [←dual_delete_dual_eq_contract, ←dual_inj_iff, dual_dual,
   delete_eq_self_iff, dual_ground]

@[simp] lemma contract_empty (M : matroid α) : M / (∅ : set α) = M :=
by { rw [contract_eq_self_iff], exact empty_disjoint _, }

lemma contract_eq_contract_inter_ground (M : matroid α) (C : set α) : M /
   C = M / (C ∩ M.E) :=
by rw [←dual_delete_dual_eq_contract, delete_eq_delete_inter_ground,
   dual_delete_dual_eq_contract,
  dual_ground]

lemma contract_eq_delete_of_subset_loops (hX : X ⊆ M.cl ∅) : M / X = M \
   X :=
begin
  rw [←dual_inj_iff, contract_dual_eq_dual_delete,
   delete_dual_eq_dual_contract,
    eq_comm, contract_eq_delete_of_subset_coloops],
  rwa dual_dual,
end

lemma basis.contract_eq_contract_delete (hI : M.basis I X) : M / X = M /
   I \ (X \ I) :=
begin
  nth_rewrite 0 ←diff_union_of_subset hI.subset,
```

```
  rw [union_comm, ←contract_contract],
  refine contract_eq_delete_of_subset_loops (λ e he, _),
  have heE : e ∈ (M / I).E, from ⟨he.2,hI.subset_ground he.1⟩,
  rw [←loop_iff_mem_cl_empty, loop_iff_not_indep heE,
   hI.indep.contract_indep_iff,
    disjoint_singleton_left, and_iff_right he.2, singleton_union],
  apply dep.not_indep _,
  rw [←hI.indep.mem_cl_iff_of_not_mem he.2, hI.cl],
  exact M.mem_cl_of_mem he.1,
end

lemma exists_eq_contract_indep_delete (M : matroid α) (C : set α) :
  ∃ (I D : set α), M.basis I (C ∩ M.E) ∧ D ⊆ (M / I).E ∧ D ⊆ C ∧ M / C =
    M / I \ D :=
begin
  obtain ⟨I, hI⟩ := M.exists_basis (C ∩ M.E),
  use [I, (C \ I) ∩ M.E, hI],
  rw [contract_ground, and_iff_right ((inter_subset_left _ _).trans
   (diff_subset _ _)),
    diff_eq, diff_eq, inter_right_comm, inter_assoc,
    and_iff_right (inter_subset_right _ _),
   contract_eq_contract_inter_ground,
    hI.contract_eq_contract_delete, diff_eq, inter_assoc],
  apply is_trans.swap,
end

lemma indep.of_contract (hI : (M / C).indep I) : M.indep I :=
begin
  obtain ⟨J, R, hJ, -, -, hM⟩ := M.exists_eq_contract_indep_delete C,
  rw [hM, delete_indep_iff, hJ.indep.contract_indep_iff] at hI,
  exact hI.1.2.subset (subset_union_left _ _),
end

@[simp] lemma contract_loop_iff_mem_cl {e : α} : (M / C).loop e ↔ e ∈
   M.cl C \ C :=
begin
  obtain ⟨I, D, hI, -, hD, hM⟩ := M.exists_eq_contract_indep_delete C,
  rw [hM, delete_loop_iff, loop_iff_dep, hI.indep.contract_dep_iff,
   disjoint_singleton_left,
    singleton_union, hI.indep.insert_dep_iff, mem_diff,
   M.cl_eq_cl_inter_ground C,
    hI.cl, and_comm (e ∉ I), and_self_right, ←mem_diff, ←mem_diff,
```

```
   diff_diff],
  apply_fun matroid.E at hM,
  rw [delete_ground, contract_ground, contract_ground,
    diff_diff, diff_eq_diff_iff_inter_eq_inter, inter_comm, inter_comm
   M.E] at hM,
  exact ⟨λ h, ⟨h.1, λ heC, h.2 (hM.subset ⟨heC, (M.cl_subset_ground _
   h.1)⟩).1⟩,
    λ h, ⟨h.1, λ h', h.2 (hM.symm.subset ⟨h', M.cl_subset_ground _ h.1 ⟩
   ).1⟩⟩,
end

@[simp] lemma contract_cl_eq (M : matroid α) (C X : set α) : (M / C).cl X
   = M.cl (X ∪ C) \ C :=
begin
  ext e,
  by_cases heX : e ∈ X,
  { by_cases he : e ∈ (M / C).E,
    { refine iff_of_true (mem_cl_of_mem' _ heX) _,
      rw [contract_ground] at he,
      exact ⟨mem_cl_of_mem' _ (or.inl heX) he.1, he.2⟩ },
    refine iff_of_false (he ∘ (λ h, cl_subset_ground _ _ h)) (he ∘ (λ h,
    _)),
    rw [contract_ground],
    exact ⟨M.cl_subset_ground _ h.1, h.2⟩ },
  suffices h' : e ∈ (M / C).cl X \ X ↔ e ∈ M.cl (X ∪ C) \ (X ∪ C),
  { rwa [mem_diff, and_iff_left heX, mem_diff, mem_union, or_iff_right
   heX, ←mem_diff ] at h' },
  rw [←contract_loop_iff_mem_cl, ←contract_loop_iff_mem_cl,
   contract_contract, union_comm],
end

lemma contract_loops_eq : (M / C).cl ∅ = M.cl C \ C :=
by simp_rw [set.ext_iff, ←loop_iff_mem_cl_empty,
   contract_loop_iff_mem_cl, iff_self,
  implies_true_iff]

lemma contract_eq_contract_iff : M / C₁ = M / C₂ ↔ C₁ ∩ M.E = C₂ ∩ M.E :=
by rw [←dual_delete_dual_eq_contract, ←dual_delete_dual_eq_contract,
   dual_inj_iff,
  delete_eq_delete_iff, dual_ground]

lemma coindep_contract_iff : (M / C).coindep X ↔ M.coindep X ∧ disjoint
```

```
   X C :=
by rw [←dual_indep_iff_coindep, contract_dual_eq_dual_delete,
   delete_indep_iff,
  dual_indep_iff_coindep]

/-- This lemma is useful where it is known (or unimportant) that `X ⊆
   M.E` -/
lemma er_contract_eq_er_contract_diff (M : matroid α) (C X : set α) :
  (M / C).er X = (M / C).er (X \ C) :=
by rw [←er_cl, contract_cl_eq, ←er_cl _ (X \ C), contract_cl_eq,
   diff_union_self]

/-- This lemma is useful where it is known (or unimportant) that `X` and `
   C` are disjoint -/
lemma er_contract_eq_er_contract_inter_ground (M : matroid α) (C X : set α
   ) :
  (M / C).er X = (M / C).er (X ∩ M.E) :=
by rw [er_eq_er_inter_ground, contract_ground,
   M.er_contract_eq_er_contract_diff _ (X ∩ M.E),
    inter_diff_assoc]

lemma basis.contract_basis_union_union (h : M.basis (J ∪ I) (X ∪ I)) (hdj
   : disjoint (J ∪ X) I)  :
  (M / I).basis J X :=
begin
  rw [disjoint_union_left] at hdj,
  have hI := h.indep.subset (subset_union_right _ _),
  simp_rw [basis, mem_maximals_set_of_iff, hI.contract_indep_iff,
   and_iff_right hdj.1,
    and_iff_right h.indep, contract_ground, subset_diff, and_iff_left
   hdj.2,
    and_iff_left ((subset_union_left _ _).trans h.subset_ground), and_imp,
    and_iff_right
      (disjoint.subset_left_of_subset_union ((subset_union_left _
   _).trans h.subset) hdj.1)],
  intros Y hYI hYi hYX hJY,
  have hu :=
    h.eq_of_subset_indep hYi (union_subset_union_left _ hJY)
   (union_subset_union_left _ hYX),
  apply_fun (λ (x : set α), x \ I) at hu,
  simp_rw [union_diff_right, hdj.1.sdiff_eq_left, hYI.sdiff_eq_left] at
   hu,
```

```
  exact hu,
end

/-- This lemma is essentially defining the 'relative rank' of `X` to `C`.
    The required set `I` can
  be obtained for any `X,C ⊆ M.E` using `M.exists_basis_union_inter_basis
    X C`. -/
lemma basis.er_contract (hI : M.basis I (X ∪ C)) (hIC : M.basis (I ∩ C)
    C) :
  (M / C).er X = (I \ C).encard :=
begin
  rw [er_contract_eq_er_contract_diff, hIC.contract_eq_contract_delete,
    delete_er_eq',
     diff_inter_self_eq_diff, basis.encard],
  apply basis.contract_basis_union_union,
  { rw [diff_union_inter, diff_diff, union_eq_self_of_subset_right
    (diff_subset _ _)],
     apply hI.basis_subset _ (union_subset_union (diff_subset _ _)
    (inter_subset_right _ _)),
     rw [union_comm, ←diff_subset_iff, subset_diff, diff_self_inter,
    diff_subset_iff, union_comm],
     exact ⟨hI.subset, disjoint_sdiff_left⟩ },
  rw [disjoint_union_left],
  exact ⟨disjoint_of_subset_right (inter_subset_right _ _)
    disjoint_sdiff_left,
     disjoint_of_subset (diff_subset _ _) (inter_subset_right _ _)
    disjoint_sdiff_left⟩,
end

lemma er_contract_add_er_eq_er_union (M : matroid α) (C X : set α) :
  (M / C).er X + M.er C = M.er (X ∪ C) :=
begin
  obtain ⟨I, D, hIC, hD, hDC, hM⟩ := M.exists_eq_contract_indep_delete C,
  obtain ⟨J, hJ, rfl⟩ :=
    hIC.exists_basis_inter_eq_of_supset (subset_union_right (X ∩ M.E) _)
    (by simp),
  rw [er_contract_eq_er_contract_inter_ground,
    contract_eq_contract_inter_ground,
     hJ.er_contract hIC, er_eq_er_inter_ground, ←hIC.encard,
    er_eq_er_inter_ground,
     inter_distrib_right, ←hJ.encard, encard_diff_add_encard_inter],
end
```

```
lemma basis.diff_subset_loops_contract (hIX : M.basis I X) : X \ I ⊆ (M /
   I).cl ∅ :=
begin
  rw [diff_subset_iff, contract_loops_eq, union_diff_self,
    union_eq_self_of_subset_left (M.subset_cl I)],
  exact hIX.subset_cl
end

noncomputable def contract_iso {β : Type*} {N : matroid β} (i : M ≃i N)
   (C : set α) :
  M / C ≃i (N / i.image C) :=
(delete_iso i.dual C).dual

end contract

section minor

variables {N M₀ M₁ M₂ : matroid α} {D : set α}

lemma contract_delete_diff (M : matroid α) (C D : set α) : M / C \ D = M
   / C \ (D \ C) :=
by rw [delete_eq_delete_iff, contract_ground, diff_eq, diff_eq, ←
   inter_inter_distrib_right,
  inter_assoc]

lemma contract_delete_comm (M : matroid α) {C D : set α} (hCD : disjoint
   C D) :
  M / C \ D = M \ D / C :=
begin
  rw [contract_eq_contract_inter_ground, (M \
   D).contract_eq_contract_inter_ground,
    delete_ground, inter_diff_distrib_left, hCD.inter_eq, diff_empty],
  obtain ⟨I, hI⟩ := M.exists_basis (C ∩ M.E),
  have hI' : (M \ D).basis I (C ∩ M.E),
  { rw delete_basis_iff, exact ⟨hI, disjoint_of_subset_left
   (inter_subset_left _ _) hCD⟩ },
  have hID : disjoint I D,
  { refine disjoint_of_subset_left hI'.subset_ground_left _, simp
   [disjoint_sdiff_left] },
  rw [hI.contract_eq_contract_delete, hI'.contract_eq_contract_delete],
  refine eq_of_indep_iff_indep_forall _ (λ J hJ, _),
```

```
  { ext, simp only [delete_delete, delete_ground, contract_ground,
   mem_diff, mem_union,
      mem_inter_iff, not_and, not_not_mem, and_imp], tauto! },

  simp only [hI.indep.contract_indep_iff, hI'.indep.contract_indep_iff,
   delete_delete,
    delete_indep_iff, disjoint_union_right, disjoint_union_left,
   and_assoc,
    and_comm _ (disjoint J D), and.congr_right_iff, iff_and_self,
   iff_true_intro hID,
    imp_true_iff],
end

lemma delete_contract_diff (M : matroid α) (D C : set α) : M \ D / C = M
   \ D / (C \ D) :=
by rw [contract_eq_contract_iff, delete_ground, diff_inter_diff_right,
   diff_eq, diff_eq,
  inter_assoc]

lemma contract_delete_contract' (M : matroid α) (C D C' : set α) :
  M / C \ D / C' = M / (C ∪ C' \ D) \ D :=
by rw [delete_contract_diff, ←contract_delete_comm _
   disjoint_sdiff_left, contract_contract]

lemma contract_delete_contract (M : matroid α) (C D C' : set α) (h :
   disjoint C' D) :
  M / C \ D / C' = M / (C ∪ C') \ D :=
by rw [contract_delete_contract', sdiff_eq_left.mpr h]

lemma contract_delete_contract_delete' (M : matroid α) (C D C' D' : set α)
   :
  M / C \ D / C' \ D' = M / (C ∪ C' \ D) \ (D ∪ D') :=
by rw [contract_delete_contract', delete_delete]

def minor (N M : matroid α) : Prop := ∃ (C ⊆ M.E) (D ⊆ M.E), disjoint C
   D ∧ N = M / C \ D

infix ` ≤m ` :75 :=  matroid.minor

lemma contract_delete_minor (M : matroid α) (C D : set α) : M / C \ D ≤m
   M :=
begin
```

```
  rw [contract_delete_diff, contract_eq_contract_inter_ground,
   delete_eq_delete_inter_ground,
    contract_ground, diff_inter_self_eq_diff, diff_inter_diff_right,
   inter_diff_right_comm],
  refine ⟨_, inter_subset_right _ _, _, inter_subset_right _ _, _, rfl⟩,
  refine disjoint_of_subset (inter_subset_left _ _) _
   (disjoint_compl_right),
  rw [diff_eq, inter_right_comm],
  exact inter_subset_right _ _,
end

instance minor_refl : is_refl (matroid α) (≤m) :=
⟨λ M, ⟨∅, empty_subset _, ∅, empty_subset _, empty_disjoint _, by simp⟩⟩

instance minor_antisymm : is_antisymm (matroid α) (≤m) :=
begin
  constructor,
  rintro M M' ⟨C,hC,D,hD,hCD,h⟩ ⟨C',hC',D',hD',hCD',h'⟩,
  have h'' := h',
  apply_fun E at h',
  simp_rw [delete_ground, contract_ground, h, delete_ground,
   contract_ground, diff_diff] at h',
  rw [eq_comm, sdiff_eq_left, disjoint_union_right, disjoint_union_right,
    disjoint_union_right] at h',
  have hC : C = ∅ := h'.1.1.1.eq_bot_of_ge hC, subst hC,
  have hD : D = ∅ := h'.1.1.2.eq_bot_of_ge hD, subst hD,
  rwa [delete_empty, contract_empty] at h,
end

instance minor_trans : is_trans (matroid α) (≤m) :=
begin
  constructor,
  rintros M₁ M₂ M₃ ⟨C₁,hC₁,D₁,hD₁,hdj,rfl⟩ ⟨C₂,hC₂,D₂,hD₂,hdj',rfl⟩,
  rw [contract_delete_contract_delete'],
  apply contract_delete_minor,
end

lemma minor.refl {M : matroid α} : M ≤m M :=
refl M

lemma minor.trans {M₁ M₂ M₃ : matroid α} (h : M₁ ≤m M₂) (h' : M₂ ≤m M₃) :
   M₁ ≤m M₃ :=
```

```
trans h h'

lemma minor.antisymm (h : N ≤m M) (h' : M ≤m N) : N = M :=
antisymm h h'

lemma contract_minor (M : matroid α) (C : set α) : M / C ≤m M :=
by { rw ←(M / C).delete_empty, apply contract_delete_minor }

lemma delete_minor (M : matroid α) (D : set α) : M \ D ≤m M :=
by { nth_rewrite 0 [←M.contract_empty], apply contract_delete_minor }

lemma minor.ground_subset (h : N ≤m M) : N.E ⊆ M.E :=
begin
  obtain ⟨C, hC, D, hD, hCD, rfl⟩ := h,
  simp only [delete_ground, contract_ground],
  exact (diff_subset _ _).trans (diff_subset _ _),
end

/-- The scum theorem. We can always realize a minor by contracting an
    independent set and deleting
   a coindependent set -/
theorem minor.exists_contract_indep_delete_coindep (h : N ≤m M) :
  ∃ C D, M.indep C ∧ M.coindep D ∧ disjoint C D ∧ N = M / C \ D :=
begin
  obtain ⟨C', hC', D', hD', hCD', rfl⟩ := h,
  obtain ⟨I, hI⟩ := M.exists_basis C',
  obtain ⟨K, hK⟩ := M*.exists_basis D',
  have hIK : disjoint I K, from disjoint_of_subset hI.subset hK.subset
   hCD',
  use [I ∪ (D' \ K), (C' \ I) ∪ K],
  refine ⟨_,_,_,_⟩,
  { have hss : D' \ K \ I ⊆ (M* / K \ I).cl ∅,
    { rw [delete_loops_eq], exact diff_subset_diff_left
   hK.diff_subset_loops_contract },
    rw [←delete_dual_eq_dual_contract, ←contract_dual_eq_dual_delete ]
   at hss,
    have hi := indep_of_subset_coloops hss,
    rw [←contract_delete_comm _ hIK, delete_indep_iff,
   hI.indep.contract_indep_iff,
      diff_union_self, union_comm] at hi,
    exact hi.1.2 },
  { rw [←dual_indep_iff_coindep],
```

```
    have hss : C' \ I \ K ⊆ (M / I \ K)**.cl ∅,
    { rw [dual_dual, delete_loops_eq], exact diff_subset_diff_left
   hI.diff_subset_loops_contract },
    have hi := indep_of_subset_coloops hss,
    rw [delete_dual_eq_dual_contract, contract_dual_eq_dual_delete,
      ←contract_delete_comm _ hIK.symm, delete_indep_iff,
   hK.indep.contract_indep_iff,
      diff_union_self] at hi,
    exact hi.1.2 },
  { rw [disjoint_union_left, disjoint_union_right, disjoint_union_right,
      and_iff_right disjoint_sdiff_right, and_iff_right hIK, and_iff_left
   disjoint_sdiff_left],
    exact disjoint_of_subset (diff_subset _ _) (diff_subset _ _) hCD'.symm
   },
  have hb : (M / C')*.basis K D',
  { rw [contract_dual_eq_dual_delete, delete_basis_iff, and_iff_right hK],
    exact hCD'.symm },
  rw [←dual_dual (M / C' \ D'), delete_dual_eq_dual_contract,
   hb.contract_eq_contract_delete,
    hI.contract_eq_contract_delete, delete_dual_eq_dual_contract,
   contract_dual_eq_dual_delete,
    dual_dual, delete_delete, contract_delete_contract],
  rw [disjoint_union_right, and_iff_left disjoint_sdiff_left],
  exact disjoint_of_subset (diff_subset _ _) (diff_subset _ _) hCD'.symm,
end

theorem minor.eq_of_ground_subset (h : N ≤m M) (hE : M.E ⊆ N.E) : M = N :
   =
begin
  obtain ⟨C, D, hC, hD, hCD, rfl⟩ :=
   h.exists_contract_indep_delete_coindep,
  simp only [delete_ground, contract_ground] at hE,
  rw [subset_diff, subset_diff] at hE,
  rw [contract_eq_contract_inter_ground, hE.1.2.symm.inter_eq,
   contract_empty,
    delete_eq_delete_inter_ground, hE.2.symm.inter_eq, delete_empty],
end

theorem minor.minor_contract_or_minor_delete {e : α} (h : N ≤m M) (he :
   e ∈ M.E \ N.E) :
    N ≤m (M / e) ∨ N ≤m (M \ e) :=
begin
```

```
  obtain ⟨C, D, hC, hD, hCD, rfl⟩ :=
   h.exists_contract_indep_delete_coindep,
  rw [delete_ground, contract_ground, diff_diff,
    diff_diff_cancel_left (union_subset hC.subset_ground
   hD.subset_ground)] at he,
  obtain (heC | heD) := he,
  { left,
    apply (delete_minor _ _).trans _,
    rw [← insert_eq_self.2 heC, ← union_singleton, union_comm, ←
   contract_contract],
    exact contract_minor _ _ },
  right,
  rw [contract_delete_comm _ hCD],
  apply (contract_minor _ _).trans _,
  rw [← insert_eq_self.2 heD, ← union_singleton, union_comm, ←
   delete_delete],
  apply delete_minor,
end

/-- An excluded minor is a minimal nonelement of S -/
def excluded_minor (S : set (matroid α)) (M : matroid α) :=
  M ∈ minimals (≤m) S^c

/-- A class is minor-closed if minors of matroids in the class are all
    in the class. -/
def minor_closed (S : set (matroid α)) : Prop := ∀ {M N}, N ≤m M → M ∈
    S → N ∈ S

lemma excluded_minor_iff (S : set (matroid α)) (hS : minor_closed S) :
  excluded_minor S M ↔ M ∉ S ∧ ∀ e ∈ M.E, M / e ∈ S ∧ M \ e ∈ S :=
begin
  rw [excluded_minor, mem_minimals_iff', mem_compl_iff,
   and.congr_right_iff],
  intro hMS,
  refine ⟨λ h e he, ⟨by_contra (fun hM', _),by_contra (fun hM', _)⟩, fun h
   N hN hNM, _⟩,
  { rw [h hM' (M.contract_minor {e}), contract_elem, contract_ground] at
   he,
    exact he.2 rfl },
  { rw [h hM' (M.delete_minor _), delete_elem, delete_ground] at he,
    exact he.2 rfl },
  refine hNM.eq_of_ground_subset (fun e heM, by_contra (fun heN, hN _)),
```

```
  obtain (h1 | h1) := hNM.minor_contract_or_minor_delete ⟨heM, heN⟩,
  { exact hS h1 (h e heM).1 },
  exact hS h1 (h e heM).2
end

lemma excluded_minor.contract_mem {S : set (matroid α)} (h :
   excluded_minor S M)
(hC : (C ∩ M.E).nonempty) : M / C ∈ S :=
begin
  by_contra hn,
  have := ((h.2 hn (contract_minor _ _)).ground_subset).antisymm
    (contract_ground_subset_ground M C),
  rw [contract_ground, eq_comm, sdiff_eq_left,
   disjoint_iff_inter_eq_empty, inter_comm] at this,
  simpa [this] using hC,
end

lemma excluded_minor.delete_mem {S : set (matroid α)} (h : excluded_minor
   S M)
(hD : (D ∩ M.E).nonempty) : M \ D ∈ S :=
begin
  by_contra hn,
  have := ((h.2 hn (delete_minor _ _)).ground_subset).antisymm
    (delete_ground_subset_ground M D),
  rw [delete_ground, eq_comm, sdiff_eq_left, disjoint_iff_inter_eq_empty,
   inter_comm] at this,
  simpa [this] using hD,
end

theorem mem_iff_no_excluded_minor_minor [M.finite] {S : set (matroid α)}
   (hS : minor_closed S) :
  M ∈ S ↔ ∀ N, excluded_minor S N → ¬(N ≤m M) :=
begin
  refine ⟨λ h N hN hNM, hN.1 (hS hNM h), λ h, by_contra (λ hMS, _)⟩,
  set minground := {X : (set α)ᵒᵈ | ∃ N, N ≤m M ∧ N.E = X ∧ N ∉ S},
  have hne : minground.nonempty := ⟨M.E, M, minor.refl, rfl, hMS⟩,
  have hfin : minground.finite,
  { refine M.ground_finite.finite_subsets.subset _,
    rintro X ⟨N, hN, rfl, h⟩,
    exact hN.ground_subset },
  obtain ⟨X, ⟨N, hN, rfl, hNS⟩, hmax⟩ := finite.exists_maximal_wrt id _
   hfin hne,
```

```
  refine h N ⟨hNS, fun M' (hM' : M' ∉ S) hM'N, _⟩ hN,
  have hNM' : N.E = M'.E := hmax M'.E ⟨M', (hM'N.trans hN), rfl, hM'⟩
   hM'N.ground_subset,
  rw [hM'N.eq_of_ground_subset hNM'.subset],
  exact minor.refl,
end

end minor

section iso_minor

/-- We have `N ≤i M` if `M` has an `N`-minor; i.e. `N` is isomorphic to a
   minor of `M` -/
def iso_minor {β : Type*} (N : matroid β) (M : matroid α) : Prop :=
  ∃ (M' : matroid α), M' ≤m M ∧ nonempty (N ≃i M')

infix ` ≤i ` :75 :=  matroid.iso_minor

lemma iso.iso_minor {β : Type*} {N : matroid β} (e : N ≃i M) : N ≤i M :=
⟨M, minor.refl, ⟨e⟩⟩

lemma minor.trans_iso {β : Type*} {N : matroid α} {M' : matroid β} (h : N
   ≤m M) (e : M ≃i M')
  : N ≤i M' :=
begin
  obtain ⟨C, hC, D, hD, hCD, rfl⟩ := h,
  set i := delete_iso (contract_iso e C) D,
  exact ⟨_,contract_delete_minor _ _ _,⟨i⟩⟩,
end

lemma minor.iso_minor {N : matroid α} (h : N ≤m M) : N ≤i M :=
⟨N, h, ⟨iso.refl N⟩⟩

lemma iso_minor.trans {α₁ α₂ α₃ : Type*} {M₁ : matroid α₁} {M₂ : matroid α₂
   }
{M₃ : matroid α₃} (h : M₁ ≤i M₂) (h' : M₂ ≤i M₃) : M₁ ≤i M₃ :=
begin
  obtain ⟨M₂', hM₂'M₃, ⟨i'⟩⟩ := h',
  obtain ⟨M₁', hM₁'M₂, ⟨i''⟩⟩ := h,
  obtain ⟨N, hN, ⟨iN⟩⟩ := hM₁'M₂.trans_iso i',
  exact ⟨N, hN.trans hM₂'M₃, ⟨i''.trans iN⟩⟩,
end
```

```
lemma iso.trans_iso_minor {β : Type*} {N' : matroid α} {N : matroid β}
  (e : N ≃i N') (h : N' ≤i M) : N ≤i M :=
e.iso_minor.trans h

end iso_minor

section update

variables [decidable_eq α] {β : Type*} {f : α → β} {s : set α} {a' : α}
   {b' : β}

lemma preimage_update  {f : α → β} (hf : f.injective) (a : α) (b : β)
   (s : set β)
  [decidable (b ∈ s)] :
  (f.update a b) ⁻¹' s = if b ∈ s then insert a (f ⁻¹' (s \ {f a})) else
   (f ⁻¹' (s \ {f a})) :=
begin
  split_ifs,
  { rw [subset_antisymm_iff, insert_subset, set.mem_preimage,
   function.update_same,
      set.preimage_diff, and_iff_right h, diff_subset_iff,
      (show {f a} = f '' {a}, by rw [image_singleton]),
      preimage_image_eq _ hf, singleton_union, insert_diff_singleton],
    refine ⟨fun x hx, _, fun x hx, _⟩,
    { obtain (rfl | hxa) := eq_or_ne x a,
      { rw [mem_preimage, function.update_same] at hx,
        apply mem_insert },
      rw [mem_preimage, function.update_noteq hxa] at hx,
      exact mem_insert_of_mem _ hx },
    obtain (rfl | hxa) := eq_or_ne x a,
    { exact mem_insert _ _ },
    rw [mem_insert_iff, mem_preimage, function.update_noteq hxa],
    exact or.inr hx },
  refine subset_antisymm (fun x hx, _) (fun x hx, _),
  { obtain (rfl | hxa) := eq_or_ne x a,
    { exact (h (by simpa using hx)).elim },
    rw [mem_preimage, function.update_noteq hxa] at hx,
    exact ⟨hx, by rwa [mem_singleton_iff, hf.eq_iff], ⟩ },
  rw [mem_preimage, mem_diff, mem_singleton_iff, hf.eq_iff] at hx,
  rw [mem_preimage, function.update_noteq hx.2],
  exact hx.1,
```

```
end

lemma pair_subset_iff {x y : α} {s : set α} : {x,y} ⊆ s ↔ x ∈ s ∧ y ∈
   s :=
  by rw [insert_subset, singleton_subset_iff]

lemma update_inj_on_iff :
  inj_on (f.update a' b') s ↔ inj_on f (s \ {a'}) ∧ (a' ∈ s → ∀ x ∈ s, f
   x = b' → x = a') :=
begin
  refine ⟨fun h, ⟨fun x hx y hy hxy, h hx.1 hy.1 _, _⟩, fun h x hx y hy
   hxy, _⟩,
  { rwa [function.update_noteq hx.2, function.update_noteq hy.2] },
  { rintro has x hxs rfl,
    exact by_contra (fun hne, hne (h hxs has (by rw [function.update_same,
      function.update_noteq hne]))) },
  obtain ⟨(rfl | hxa), (rfl | hya)⟩ := ⟨eq_or_ne x a', eq_or_ne y a'⟩,
  { refl },
  { rw [function.update_same, function.update_noteq hya, eq_comm] at hxy,
    rw [h.2 hx y hy hxy] },
  { rw [function.update_same, function.update_noteq hxa] at hxy,
    rw [h.2 hy x hx hxy] },
  rwa [function.update_noteq hxa, function.update_noteq hya, h.1.eq_iff ⟨
   hx, hxa⟩ ⟨hy,hya⟩] at hxy
end

@[simp] lemma image_update (a : α) (f : α → β) (s : set α) (b : β)
   [decidable (a ∈ s)] :
  (f.update a b) '' s = if a ∈ s then insert b (f '' (s \ {a})) else f '' s :
   =
begin
  split_ifs,
  { rw [subset_antisymm_iff, image_subset_iff],
    refine ⟨λ x hxs, (em (x = a)).elim (fun heq, _) (fun hne, or.inr _), λ
     x, _⟩,
    { rw [mem_preimage, function.update_apply, if_pos heq]; exact
   mem_insert _ _ },
    { exact ⟨x, ⟨hxs, hne⟩, by rw [function.update_noteq hne]⟩ },
    rintro (rfl | ⟨x, hx, rfl⟩),
    { use a, simpa },
    exact ⟨x, hx.1, function.update_noteq hx.2 _ _⟩ },
  rw [subset_antisymm_iff, image_subset_iff, image_subset_iff],
```

```
  refine ⟨fun x hxs, ⟨x, hxs, _⟩, fun x hxs, ⟨x, hxs, _⟩⟩;
  { rw [function.update_noteq], exact fun hxa, h (by rwa ← hxa) },
end

@[simp] lemma update_id_inj_on_iff {a b : α} :
    inj_on ((@id α).update a b) s ↔ (a ∈ s → b ∈ s → a = b) :=
begin
  rw [update_inj_on_iff, and_iff_right (function.injective_id.inj_on _)],
  refine ⟨fun h has hbs, (h has b hbs rfl).symm, _⟩,
  rintro h has _ hbs rfl,
  exact (h has hbs).symm,
end

end update

section bij_on

/-- If `f` maps `s` bijectively to `t` and, then for any `s ⊆ s₁` and `t ⊆ t'
   ⊆ f '' s₁`,
  there is some `s ⊆ s' ⊆ s₁` so that `f` maps `s'` bijectively to `t'`. -/
theorem set.bij_on.extend_of_subset {β : Type*} {f : α → β} {s s₁ : set
   α} {t t' : set β}
    (h : bij_on f s t) (hss₁ : s ⊆ s₁) (htt' : t ⊆ t') (ht' : t' ⊆ f '' s₁) :
    ∃ s', s ⊆ s' ∧ s' ⊆ s₁ ∧ bij_on f s' t' :=
begin
  have hex : ∀ (b : (t' \ t)),  ∃ a, a ∈ s₁ \ s ∧ f a = b,
  { rintro ⟨b, hb⟩,
    obtain ⟨a, ha, rfl⟩ := ht' hb.1,
    exact ⟨_, ⟨ha, fun has, hb.2 (h.maps_to has)⟩, rfl⟩ },
  choose g hg using hex,
  have hinj : inj_on f (s ∪ range g),
  { rw [inj_on_union, and_iff_right h.inj_on, and_iff_left],
    { rintro _ ⟨⟨x,hx⟩, rfl⟩ _ ⟨⟨x', hx'⟩,rfl⟩ hf,
      simp only [(hg _).2, (hg _).2] at hf,
      rw [subtype.coe_mk, subtype.coe_mk] at hf,
      subst hf },
    { rintro x hx _ ⟨⟨y,hy⟩, hy', rfl⟩ h_eq,
      rw [(hg _).2] at h_eq,
      obtain (rfl : _ = y) := h_eq,
      exact hy.2 (h.maps_to hx), },
    rw [disjoint_iff_forall_ne],
    rintro x hx _ ⟨y, hy, rfl⟩ rfl,
```

```
    have h' := h.maps_to hx,
    rw [(hg _).2] at h',
    exact y.prop.2 h' },
  have hp : bij_on f (range g) (t' \ t),
  { apply bij_on.mk,
    { rintro _ ⟨x, hx, rfl⟩; rw [(hg _).2]; exact x.2},
    { exact hinj.mono (subset_union_right _ _), },
    exact fun x hx, ⟨g ⟨x,hx⟩, by simp [(hg _).2] ⟩,},
  refine ⟨s ∪ range g, subset_union_left _ _, union_subset hss₁  _, _⟩,
  { rintro _ ⟨x, hx, rfl⟩; exact (hg _).1.1 },
  convert h.union hp _,
  { exact (union_diff_cancel htt').symm },
  exact hinj,
end

theorem set.bij_on.extend {β : Type*} {f : α → β} {s : set α} {t t' :
   set β} (h : bij_on f s t)
    (htt' : t ⊆ t') (ht' : t' ⊆ range f) : ∃ s', s ⊆ s' ∧ bij_on f s' t' :=
    by
  simpa using h.extend_of_subset (subset_univ s) htt' (by simpa)

end bij_on

section restrict

def restrict' (M : matroid α) (X : set α) : matroid α :=
matroid_of_indep X (λ I, M.indep I ∧ I ⊆ X ∩ M.E) ⟨M.empty_indep,
   empty_subset _⟩
(λ I J hJ hIJ, ⟨hJ.1.subset hIJ, hIJ.trans hJ.2⟩)
(begin
  set Y := X ∩ M.E with hY_def,
  have hY : Y ⊆ M.E := inter_subset_right _ _,
  rintro I I' ⟨hI, hIY⟩ hIn hI',
  rw ←basis_iff_mem_maximals at hIn hI',
  obtain ⟨B', hB', rfl⟩ := hI'.exists_base,
  obtain ⟨B, hB, hIB, hBIB'⟩ := hI.exists_base_subset_union_base hB',

  rw [hB'.inter_basis_iff_compl_inter_basis_dual hY, diff_inter_diff] at
   hI',

  have hss : M.E \ (B' ∪ Y) ⊆ M.E \ (B ∪ Y),
  { rw [subset_diff, and_iff_right (diff_subset _ _), ←
```

```
    subset_compl_iff_disjoint_left,
       diff_eq, compl_inter, compl_compl, ←union_assoc, union_subset_iff,
       and_iff_left (subset_union_right _ _)],
     refine hBIB'.trans (union_subset (hIY.trans _)
       (subset_union_of_subset_left (subset_union_right _ _) _)),
     apply subset_union_right },

  have hi : M*.indep (M.E \ (B ∪ Y)),
  { rw [dual_indep_iff_coindep, coindep_iff_exists],
    exact ⟨B, hB, disjoint_of_subset_right (subset_union_left _ _)
    disjoint_sdiff_left⟩ },
  have h_eq := hI'.eq_of_subset_indep hi hss
    (by {rw [diff_subset_iff, union_assoc, union_diff_self, ←
    union_assoc], simp }),

  rw [h_eq, ←diff_inter_diff, ←
    hB.inter_basis_iff_compl_inter_basis_dual hY] at hI',

  have hssu : I ⊂ (B ∩ Y) := (subset_inter hIB hIY).ssubset_of_ne
    (by { rintro rfl, exact hIn hI' }),

  obtain ⟨e, heBY, heI⟩ := exists_of_ssubset hssu,
  exact ⟨e, ⟨⟨(hBIB' heBY.1).elim (λ h', (heI h').elim) id ,heBY.2⟩,heI⟩,
    (hB.indep.inter_right Y).subset (insert_subset.mpr ⟨heBY,hssu.subset⟩),
    insert_subset.mpr ⟨heBY.2,hssu.subset.trans (inter_subset_right _ _)⟩⟩,
end)
(begin
  rintro X hX I ⟨hI, hIX⟩ hIA,
  obtain ⟨J, hJ, hIJ⟩ := hI.subset_basis_of_subset (subset_inter hIA hIX),
  refine ⟨J, ⟨⟨hJ.indep,hJ.subset.trans (inter_subset_right _ _)⟩,hIJ,
    hJ.subset.trans (inter_subset_left _ _)⟩, λ B hB hJB, _⟩,
  rw hJ.eq_of_subset_indep hB.1.1 hJB (subset_inter hB.2.2 hB.1.2),
end)
(fun I hI, hI.2.trans (inter_subset_left _ _))

@[simp] lemma restrict'_indep_iff {M : matroid α} {X I : set α} :
  (M.restrict' X).indep I ↔ M.indep I ∧ I ⊆ X :=
begin
  simp only [restrict', subset_inter_iff, matroid_of_indep_apply,
   and.congr_right_iff,
    and_iff_left_iff_imp],
  exact fun h _, h.subset_ground
```

```
end

end restrict

section preimage

/-- The pullback of a matroid on `β` by a function `f : α → β` to a
   matroid on `α`.
  Elements with the same image are parallel and the ground set is `f ⁻¹' M.E`. -/
def preimage {β : Type*} (M : matroid β) (f : α → β) : matroid α :=
   matroid_of_indep
  (f ⁻¹' M.E) (fun I, M.indep (f '' I) ∧ inj_on f I) ( by simp )
  ( fun I J ⟨h, h'⟩ hIJ, ⟨h.subset (image_subset _ hIJ), inj_on.mono hIJ
   h'⟩ )
  ( begin
    rintro I B ⟨hI, hIinj⟩ hImax hBmax,
    change I ∉ maximals _ {I : set α  | _} at hImax,
    change B ∈ maximals _ {I : set α  | _} at hBmax,
    simp only [mem_maximals_iff', mem_set_of_eq, hI, hIinj, and_self,
   and_imp,
      true_and, not_forall, exists_prop, not_and] at hImax hBmax,

    obtain ⟨I', hI', hI'inj, hII', hne⟩ := hImax,

    have h₁ : ¬(M.restrict' (set.range f)).base (f '' I),
    { refine fun hB, hne _,
      have h_im := hB.eq_of_subset_indep _ (image_subset _ hII'),
      { refine hII'.antisymm (fun x hxI', _),
        rw [← hI'inj.mem_image_iff hII' hxI', h_im],
        exact mem_image_of_mem _ hxI' },
      rwa [restrict'_indep_iff, and_iff_left (image_subset_range _ _)] },


    have h₂ : (M.restrict' (range f)).base (f '' B),
    { refine indep.base_of_maximal (by simpa using hBmax.1.1) (fun J hJi
   hBJ, _),
      simp only [restrict'_indep_iff] at hJi,
      obtain ⟨J₀, hBJ₀, hJ₀⟩ := hBmax.1.2.bij_on_image.extend hBJ hJi.2,
      obtain rfl := hJ₀.image_eq,
      rw [hBmax.2 hJi.1 hJ₀.inj_on hBJ₀]},
```

```
    obtain ⟨_, ⟨⟨e, he, rfl⟩, he'⟩, hei⟩ := indep.exists_insert_of_not_base
   (by simpa) h₁ h₂,
    have heI : e ∉ I := fun heI, he' (mem_image_of_mem f heI),
    rw [← image_insert_eq, restrict'_indep_iff] at hei,
    exact ⟨e, ⟨he, heI⟩, hei.1, (inj_on_insert heI).2 ⟨hIinj, he'⟩⟩,
  end )
  ( begin

    rintro X - I ⟨hI, hIinj⟩ hIX,
    obtain ⟨J, hJ, hIJ⟩ :=
      (show (M.restrict' (range f)).indep (f '' I), by
   simpa).subset_basis_of_subset
      (image_subset _ hIX) (image_subset_range _ _),

    simp only [basis_iff, restrict'_indep_iff] at hJ,

    obtain ⟨J₀, hIJ₀, hJ₀X, hbij⟩ := hIinj.bij_on_image.extend_of_subset
   hIX hIJ hJ.2.1,
    use J₀,
    simp only [mem_maximals_iff', mem_set_of_eq],
    rw [and_iff_left hbij.inj_on, hbij.image_eq, and_iff_right hJ.1.1,
   and_iff_right hIJ₀,
      and_iff_right hJ₀X],
    rintro K ⟨⟨hK, hKinj⟩, hIK, hKX⟩ hJ₀K,
    obtain rfl := hJ.2.2 _ ⟨hK, image_subset_range _ _⟩ _ (image_subset _
   hKX),
    { refine hJ₀K.antisymm (fun x hxK, _),
      rwa [← hKinj.mem_image_iff hJ₀K hxK, hbij.image_eq,
   hKinj.mem_image_iff subset.rfl hxK] },
    rw [← hbij.image_eq ],
    exact image_subset _ hJ₀K,
  end )
  (  fun I hI e heI, hI.1.subset_ground ⟨e, heI, rfl⟩  )

@[simp] lemma preimage_ground_eq {β : Type*} (M : matroid β) (f : α → β
   ) :
  (M.preimage f).E = f ⁻¹' M.E := rfl

@[simp] lemma preimage_indep_iff {β : Type*} {M : matroid β} {f : α → β
   } {I : set α} :
    (M.preimage f).indep I ↔ M.indep (f '' I) ∧ inj_on f I :=
by simp [preimage]
```

```
end preimage

section single_extensions

def add_loop (M : matroid α) (f : α) : matroid α := M.restrict' (insert f
   M.E)

@[simp] lemma add_loop_ground (M : matroid α) (f : α) : (M.add_loop f).E =
    insert f M.E := rfl

@[simp] lemma add_loop_indep_iff {f : α} : (M.add_loop f).indep I ↔
   M.indep I :=
begin
  rw [add_loop, restrict'_indep_iff, and_iff_left_iff_imp],
  exact fun hI, hI.subset_ground.trans (subset_insert _ _),
end

lemma eq_add_loop_iff {f : α} (M M' : matroid α) (hf : f ∉ M.E) :
    M' = add_loop M f ↔ M'.loop f ∧ M' \ f = M :=
begin
  rw [loop_iff_dep, dep_iff, singleton_subset_iff],
  split,
  { rintro rfl,
    rw [add_loop_indep_iff, add_loop_ground, and_iff_left (mem_insert _
   _), indep_singleton,
      and_iff_right (show ¬M.nonloop f, from fun h, hf h.mem_ground),
      eq_iff_indep_iff_indep_forall, delete_elem, delete_ground,
   add_loop_ground],
    simp only [insert_diff_of_mem, mem_singleton, sdiff_eq_left,
   disjoint_singleton_right,
      delete_indep_iff, add_loop_indep_iff, and_iff_left_iff_imp,
   and_iff_right hf],
    rintro I hI - hfI,
    exact (hI hfI).2 rfl },
  rintro ⟨hfl,rfl⟩,
  apply eq_of_indep_iff_indep_forall _ (fun I hI, _),
  { simp only [delete_elem, add_loop_ground, delete_ground,
   insert_diff_singleton],
    rw [insert_eq_of_mem hfl.2] },
  simp only [delete_elem, add_loop_indep_iff, delete_indep_iff,
   disjoint_singleton_right,
```

```
    iff_self_and],
  exact fun hI' hfI, hfl.1 (hI'.subset (singleton_subset_iff.2 hfI)),
end

def add_coloop (M : matroid α) (f : α) : matroid α := (M*.add_loop f)*

lemma add_coloop_eq {f : α} (M M' : matroid α) (hf : f ∉ M.E) :
    M' = add_coloop M f ↔ M'.coloop f ∧ M' \ f = M :=
begin
  rw [add_coloop, eq_dual_comm, eq_comm, eq_add_loop_iff _ _ (show f ∉
   M*.E, from hf),
    dual_loop_iff_coloop, eq_dual_comm, delete_elem,
   dual_delete_dual_eq_contract,
    delete_elem, and.congr_right_iff, eq_comm],
  intro hf',
  rw [contract_eq_delete_of_subset_coloops],
  rwa [singleton_subset_iff, ← coloop_iff_mem_cl_empty ],
end

lemma add_coloop_del_eq {f : α} (M : matroid α) (hf : f ∉ M.E) :
   add_coloop M f \ f = M :=
  (((M.add_coloop_eq _) hf).1 rfl).2

@[simp] lemma add_coloop_ground (M : matroid α) (f : α) : (M.add_coloop
   f).E = insert f M.E := rfl

variables {e f : α} [decidable_eq α]

/-- extend `e` in `M` by a parallel element `f`. -/
def parallel_extend (M : matroid α) (e f : α) : matroid α := M.preimage
   ((@id α).update f e)

lemma parallel_extend_ground (he : e ∈ M.E) (f : α) : (M.parallel_extend
   e f).E = insert f M.E :=
begin
  simp only [parallel_extend, matroid.preimage_ground_eq],
  refine subset_antisymm _ (insert_subset.2 ⟨by simpa, fun x hx, _⟩),
  { rintro x hx,
    simp only [set.mem_preimage] at hx,
    obtain (rfl | hx') := eq_or_ne x f,
    { exact mem_insert _ _ },
    rw [function.update_apply, if_neg hx'] at hx,
```

```
    exact or.inr hx },
  obtain (rfl | hx') := eq_or_ne x f,
  { simpa },
  rwa [mem_preimage, function.update_apply, if_neg hx'],
end

/-- If `e ∉ M.E`, then `M.parallel_extend e f` has the junk value `M \ f`.
    -/
lemma parallel_extend_not_mem (he : e ∉ M.E) (f : α) : M.parallel_extend
    e f = M \ f :=
begin
  classical,
  simp_rw [eq_iff_indep_iff_indep_forall, parallel_extend,
   preimage_ground_eq,
    preimage_update function.injective_id, if_neg he, preimage_id,
   preimage_indep_iff, image_update,
    image_id, id, delete_elem, delete_indep_iff, delete_ground,
   and_iff_right rfl, subset_diff,
    disjoint_singleton_right, update_id_inj_on_iff],
  rintro I ⟨hI, hfI⟩,
  simp [if_neg hfI, hfI],
end

lemma parallel_extend_delete_eq (M : matroid α) (e f : α) (hf : f ∉ M.E)
    :
    (M.parallel_extend e f) \ f = M :=
begin
  classical,
  obtain (he | he) := em' (e ∈ M.E),
  { rwa [parallel_extend_not_mem he, delete_elem, delete_elem,
   delete_delete, union_singleton,
      pair_eq_singleton, delete_eq_self_iff, disjoint_singleton_left],  },
  simp_rw [delete_elem, eq_iff_indep_iff_indep_forall, delete_ground,
   parallel_extend_ground he,
    delete_indep_iff, subset_diff, disjoint_singleton_right,
   insert_diff_self_of_not_mem hf,
    and_iff_right rfl, parallel_extend, preimage_indep_iff, image_update,
    update_inj_on_iff, and_iff_right (function.injective_id.inj_on _)],
  rintro I ⟨hIs, hfI⟩,
  simp [if_neg hfI, hfI],
end
```

```
lemma parallel_extend_indep_iff (he : M.nonloop e) (hf : f ∉ M.E) {I :
   set α} :
    (M.parallel_extend e f).indep I ↔
      (f ∉ I ∧ M.indep I) ∨ (f ∈ I ∧ e ∉ I ∧ M.indep (insert e (I \
   {f}))) :=
begin
  classical,
  rw [parallel_extend, preimage_indep_iff, update_inj_on_iff,
    and_iff_right (function.injective_id.inj_on _), image_update,
   image_id, image_id],
  split_ifs,
  { rw [and_iff_right h, iff_true_intro h, true_implies_iff, not_true,
   false_and, false_or,
      and_comm, and.congr_left_iff],
    refine fun h, ⟨fun h' heI, hf _, fun h' x hxI, _⟩,
    { rw [← h' _ heI rfl], exact he.mem_ground },
    rintro rfl,
    exact (h' hxI).elim },
  rw [iff_false_intro h, false_implies_iff, and_true, ← not_true,
   not_not, true_and, not_true,
    false_and, or_false],
end

lemma parallel_extend_circuit (he : M.nonloop e) (hf : f ∉ M.E) :
    (M.parallel_extend e f).circuit {e,f} :=
begin
  simp_rw [circuit_iff_dep_forall_diff_singleton_indep, dep_iff,
    parallel_extend_ground he.mem_ground, insert_subset,
    and_iff_right (mem_insert_of_mem _ he.mem_ground),
    and_iff_left (singleton_subset_iff.2 (mem_insert _ _)),
   mem_insert_iff, mem_singleton_iff,
    parallel_extend_indep_iff he hf],
  simp only [mem_insert_iff, or_true, not_true, false_and,
   eq_self_iff_true, mem_singleton_iff,
  true_or, not_false_iff, mem_diff, true_and, not_not, forall_eq_or_imp,
   insert_diff_of_mem,
  sdiff_sdiff_self, bot_eq_empty, insert_emptyc_eq, indep_singleton,
   forall_eq, or_false, he,
    and_true],
  obtain (rfl | hne) := eq_or_ne e f,
  { simp },
  simp only [hne.symm, false_and, not_false_iff, false_or, true_and],
```

```
  rwa [← insert_diff_singleton_comm hne, diff_self, insert_emptyc_eq,
   indep_singleton],
end

lemma eq_parallel_extend_iff {M M' : matroid α} (he : M.nonloop e) (hf :
   f ∉ M.E) :
    M' = M.parallel_extend e f ↔ M'.circuit {e,f} ∧ M' \ f = M :=
begin
  split,
  { rintro rfl,
    exact ⟨parallel_extend_circuit he hf, M.parallel_extend_delete_eq e f
   hf⟩ },
  rintro ⟨hC, rfl⟩,
  have hef := pair_subset_iff.1 hC.subset_ground,
  have hne : e ≠ f := by {rintro rfl, exact hf he.mem_ground},
  have heE : e ∈ (M' \ f).E,
  { rw [delete_elem, delete_ground, mem_diff], exact ⟨hef.1, hne⟩ },
  rw [eq_iff_indep_iff_indep_forall, parallel_extend_ground heE,
   delete_elem, delete_ground,
    insert_diff_singleton, eq_comm, insert_eq_self, and_iff_right hef.2],
   swap,

  simp_rw [←delete_elem, parallel_extend_indep_iff he hf, delete_elem,
   delete_indep_iff,
    disjoint_singleton_right],
  simp only [mem_insert_iff, not_mem_diff_singleton, or_false,
    and_iff_left (show ¬ (f = e) , from hne.symm)],
  refine fun I hI, _,
  obtain (hfI | hfI) := em (f ∈ I),
  { simp only [hfI, not_true, and_false, true_and, false_or],
    refine ⟨λ h, ⟨fun heI, _, _⟩, λ h, _⟩,
    { exact hC.dep.not_indep (h.subset (pair_subset_iff.2 ⟨heI, hfI⟩)) },
    { rw [indep_iff_forall_subset_not_circuit', insert_subset,
   and_iff_right hef.1,
        and_iff_left ((diff_subset _ _).trans hI)],
      rintro C' hC'ss hC',
      have hCne : C' ≠ {e,f},
      { rintro rfl, obtain (rfl | h') := hC'ss (or.inr rfl), exact hne
   rfl, exact h'.2 rfl },
      obtain ⟨C₀, hC₀ss, hC₀⟩ := hC'.elimination hC hCne e,
      refine hC₀.dep.not_indep (h.subset (hC₀ss.trans _)),
      rw [diff_subset_iff, union_subset_iff, insert_subset,
```

```
        and_iff_right (show e ∈ {e} ∪ I, from or.inl rfl),
    singleton_subset_iff,
        and_iff_left (show f ∈ {e} ∪ I, from or.inr hfI),
    singleton_union] ,
      exact hC'ss.trans (insert_subset_insert (diff_subset _ _)) },
    rw [indep_iff_forall_subset_not_circuit],
    rintro C' hC'ss hC',
    have hCne : C' ≠ {e,f},
    { rintro rfl, exact h.1 (pair_subset_iff.1 hC'ss).1, },
    obtain ⟨C₀, hC₀ss, hC₀⟩ := hC'.elimination hC hCne f,
    rw [union_insert, union_singleton, insert_comm, insert_diff_of_mem _
    (by simp : f ∈ {f})]
      at hC₀ss,

    refine hC₀.dep.not_indep (h.2.subset (hC₀ss.trans _)),
    rw [insert_diff_singleton_comm hne],
    exact (diff_subset_diff_left (insert_subset_insert hC'ss)) },
  simp [hfI],
end

/-- extend `e` in `M` by an element `f` in series. -/
def series_extend (M : matroid α) (e f : α) : matroid α :=
   (M✶.parallel_extend e f)✶

lemma series_extend_ground (he : e ∈ M.E) : (M.series_extend e f).E =
   insert f M.E :=
by simp [series_extend, parallel_extend_ground (show e ∈ M✶.E, from he)]

lemma series_extend_contract_eq (M : matroid α) (e f : α) (hf : f ∉ M.E)
   :
  (M.series_extend e f) / f = M :=
dual_inj
  (by rwa [series_extend, contract_elem, dual_contract_dual_eq_delete, ←
   delete_elem,
    parallel_extend_delete_eq ])

lemma series_extend_cocircuit (heE : e ∈ M.E) (he : ¬ M.coloop e) (hf : f
   ∉ M.E) :
  (M.series_extend e f).cocircuit {e,f} :=
begin
  have hnl : M✶.nonloop e,
  { rw [nonloop, dual_loop_iff_coloop], exact ⟨he, heE⟩ },
```

```
  rw [←dual_circuit_iff_cocircuit ],
  convert parallel_extend_circuit hnl hf,
  rw [eq_comm, eq_dual_comm],
  refl,
end

lemma eq_series_extend_iff {M M' : matroid α} (heE : e ∈ M.E) (he : ¬
   M.coloop e) (hf : f ∉ M.E) :
  M' = M.series_extend e f ↔ M'.cocircuit {e,f} ∧ M' / f = M :=
begin
  have hnl : M*.nonloop e,
  { rw [nonloop, dual_loop_iff_coloop], exact ⟨he, heE⟩ },
  rw [series_extend, ← dual_circuit_iff_cocircuit, ← dual_inj_iff,
   dual_dual,
    eq_parallel_extend_iff hnl (show f ∉ M*.E, from hf), eq_dual_comm,
   delete_elem,
    dual_delete_dual_eq_contract, ← contract_elem, eq_comm],
end

end single_extensions


section unif

/-- Given `I ⊆ E`, the matroid on `E` whose unique base is the set `I`.
  (If `I` is not a subset of `E`, the base is `I ∩ E` )-/
def trivial_on (I E : set α) : matroid α :=
matroid_of_base E (λ X, X = I ∩ E) ⟨_, rfl⟩
(by { rintro B₁ B₂ rfl rfl x h, simpa using h })
(begin
  rintro Y - J ⟨B, rfl, hJB⟩ hJY,
  use I ∩ Y ∩ E,
  simp only [mem_maximals_set_of_iff, exists_eq_left, subset_inter_iff,
   inter_subset_right,
    and_true, and_imp],
  rw [inter_right_comm, and_iff_left (inter_subset_right _ _),
   inter_assoc,
    and_iff_right (inter_subset_left _ _), and_iff_left hJY,
    and_iff_right (subset_inter_iff.mp hJB)],
  exact λ X hXI hXE hJX hXY hss, hss.antisymm (subset_inter hXI
   (subset_inter hXE hXY)),
end)
```

```
(by { rintro B rfl, apply inter_subset_right })

lemma trivial_on_base_iff (hIE : I ⊆ E) : (trivial_on I E).base B ↔ B =
   I :=
by simp only [trivial_on, matroid_of_base_apply,
   inter_eq_self_of_subset_left hIE]

lemma trivial_on_indep_iff (hIE : I ⊆ E) : (trivial_on I E).indep J ↔ J
   ⊆ I :=
by { simp_rw [indep_iff_subset_base, trivial_on_base_iff hIE], simp }

/-- The truncation of a matroid to finite rank `k`. The independent sets
   of the truncation
  are the independent sets of the matroid of size at most `k`. -/
def truncate (M : matroid α) (k : ℕ) : matroid α :=
matroid_of_indep_of_bdd' M.E (λ I, M.indep I ∧ I.finite ∧ I.ncard ≤ k)
(by simp)
(λ I J h hIJ, ⟨h.1.subset hIJ, h.2.1.subset hIJ, (ncard_le_of_subset hIJ
   h.2.1).trans h.2.2⟩ )
(begin
  rintro I J ⟨hI, hIfin, hIc⟩ ⟨hJ, hJfin, hJc⟩ hIJ,
  obtain ⟨e, heJ, heI, hi⟩ := hI.augment_of_finite hJ hIfin hIJ,
  refine ⟨e, heJ, heI, hi, hIfin.insert e, (ncard_insert_le _ _).trans _⟩,
  rw [nat.add_one_le_iff],
  exact hIJ.trans_le hJc,
end)
(⟨k, λ I, and.right⟩)
(λ I hI, hI.1.subset_ground)

@[simp] lemma truncate_indep_iff : (M.truncate k).indep I ↔ (M.indep I ∧
   I.finite ∧ I.ncard ≤ k) :=
by simp [truncate]

lemma truncate_base_iff [finite_rk M] (h : k ≤ M.rk) :
  (M.truncate k).base B ↔ M.indep B ∧ B.ncard = k :=
begin
  simp_rw [base_iff_maximal_indep, truncate_indep_iff, and_imp],
  split,
  { rintro ⟨⟨hBi, hBin, hBc⟩, hBmax⟩,
    refine ⟨hBi, hBc.antisymm _⟩,
    obtain ⟨B', hB', hBB'⟩ := hBi.exists_base_supset,
    rw ←hB'.card at h,
```

```
    obtain ⟨J, hBJ, hJB', rfl⟩ := exists_intermediate_set' hBc h hBB',
    rw hBmax J (hB'.indep.subset hJB') (hB'.finite.subset hJB') rfl.le hBJ
   },
  rintro ⟨hB, rfl⟩,
  exact ⟨⟨hB, hB.finite, rfl.le⟩, λ I hI hIfin hIc hBI,
   eq_of_subset_of_ncard_le hBI hIc hIfin⟩,
end

/-- The matroid on `E` whose only basis is `E` -/
def free_on (E : set α) : matroid α := trivial_on E E

@[simp] lemma free_on_base_iff (E : set α) : (free_on E).base B ↔ B = E :
   =
by rw [free_on, trivial_on_base_iff subset.rfl]

@[simp] lemma free_on_indep_iff (E : set α) : (free_on E).indep I ↔ I ⊆
   E :=
by rw [free_on, trivial_on_indep_iff subset.rfl]

@[simp] lemma free_on_rk_eq (E : set α) : (free_on E).rk = E.ncard :=
by { obtain ⟨B, hB⟩ := (free_on E).exists_base, rw [←hB.card,
   (free_on_base_iff _).mp hB] }

/-- A uniform matroid with a given rank and ground set -/
def set.unif_on (E : set α) (k : ℕ) := (free_on E).truncate k

@[simp] lemma set.unif_on_ground_eq : (E.unif_on k).E = E := rfl

@[simp] lemma set.unif_on_indep_iff : (E.unif_on k).indep I ↔ I.finite ∧
   I.ncard ≤ k ∧ I ⊆ E :=
by {simp [set.unif_on, and_comm (I ⊆ E), and_assoc], }

lemma set.unif_on_indep_iff' : (E.unif_on k).indep I ↔ I.encard ≤ k ∧ I
   ⊆ E :=
by rw [encard_le_coe_iff, set.unif_on_indep_iff, and_assoc]

lemma set.eq_unif_on_iff : M = E.unif_on a ↔ M.E = E ∧ ∀ I, M.indep I ↔
   I.encard ≤ a ∧ I ⊆ E :=
begin
  simp_rw [eq_iff_indep_iff_indep_forall, set.unif_on_ground_eq,
   set.unif_on_indep_iff',
    and.congr_right_iff],
```

```
  rintro rfl,
  exact ⟨λ h I, ⟨λ hI, (h I hI.subset_ground).mp hI, λ hI, (h I hI.2).mpr
   hI⟩,
    λ h I hIE, h I⟩,
end

/-- A uniform matroid of a given rank whose ground set is the universe
   of a type -/
def unif_on (α : Type*) (k : ℕ) := (univ : set α).unif_on k

@[simp] lemma unif_on_indep_iff [_root_.finite α] : (unif_on α k).indep I
   ↔ I.ncard ≤ k :=
by simp only [unif_on, set.unif_on_indep_iff, subset_univ, and_true,
   and_iff_right_iff_imp,
    iff_true_intro (to_finite I), imp_true_iff]

/-- A canonical uniform matroid, with rank `a` and ground type `fin b`. -/
def unif (a b : ℕ) := unif_on (fin b) a

@[simp] lemma unif_ground_eq (a b : ℕ) : (unif a b).E = univ := rfl

@[simp] lemma unif_indep_iff (I : set (fin b)) : (unif a b).indep I ↔
   I.ncard ≤ a :=
by rw [unif, unif_on_indep_iff]

@[simp] lemma unif_indep_iff' (I : set (fin b)) : (unif a b).indep I ↔
   I.encard ≤ a :=
by rw [unif_indep_iff, encard_le_coe_iff, and_iff_right (to_finite I)]

@[simp] lemma unif_base_iff (hab : a ≤ b) {B : set (fin b)} :
  (unif a b).base B ↔ B.ncard = a :=
begin
  simp only [unif, unif_on, set.unif_on],
  rw [truncate_base_iff, free_on_indep_iff, and_iff_right (subset_univ
   _)],
  rwa [free_on_rk_eq, ncard_eq_to_finset_card, finite.to_finset_univ,
   finset.card_fin],
end

lemma iso_unif_iff {a b : ℕ} {M : matroid α} :
  nonempty (M ≃i (unif a b)) ↔ (M = M.E.unif_on a ∧ M.E.encard = (b : ℕ
   ∞)) :=
```

```
begin
  refine ⟨λ h, _, λ h, _⟩,
  { obtain ⟨i⟩ := h,
    set e := i.to_equiv,
    rw [encard, part_enat.card_congr e, unif_ground_eq,
   part_enat.card_eq_coe_fintype_card,
      part_enat.with_top_equiv_coe, nat.cast_inj, ←set.to_finset_card,
   to_finset_univ,
      finset.card_fin, eq_self_iff_true, and_true,
   eq_iff_indep_iff_indep_forall,
      set.unif_on_ground_eq, and_iff_right rfl],
    intros I hI,
    rw [set.unif_on_indep_iff, and_iff_left hI, ←encard_le_coe_iff,
   i.on_indep hI, unif_indep_iff',
      iso.image, encard_image_of_injective _ (subtype.coe_injective),
      encard_image_of_injective _ (equiv_like.injective i),
      encard_preimage_of_injective_subset_range subtype.coe_injective],
    rwa subtype.range_coe },
  rw [encard_eq_coe_iff, ncard] at h,
  obtain ⟨h1, hfin, h'⟩ := h,
  haveI := finite_coe_iff.mpr hfin,
  set e := (finite.equiv_fin_of_card_eq h').trans (equiv.set.univ (fin
   b)).symm,
  refine ⟨@iso_of_indep _ _ M (unif a b) e (λ I, _)⟩,
  apply_fun indep at h1,
  rw [h1, set.unif_on_indep_iff'],
  simp only [image_subset_iff, subtype.coe_preimage_self, subset_univ,
   and_true, equiv.coe_trans,
    function.comp_app, equiv.set.univ_symm_apply, unif_indep_iff',
    encard_image_of_injective _ subtype.coe_injective],
  rw [encard_image_of_injective],
  intros x y,
  simp,
end

/-- Horrible proof. Should be improved using `simple` api -/
lemma iso_line_iff {k : ℕ} (hk : 2 ≤ k) :
  nonempty (M ≃i (unif 2 k)) ↔
    (∀ e f ∈ M.E, M.indep {e,f}) ∧ M.rk = 2 ∧ M.E.finite ∧ M.E.ncard = k
   :=
begin
  simp_rw [iso_unif_iff, encard_eq_coe_iff, ← and_assoc,
```

```
   and.congr_left_iff,
    set.eq_unif_on_iff, and_iff_right rfl, nat.cast_bit0, enat.coe_one],
  rintro rfl hfin,
  have lem : ∀ x y, ({x,y} : set α).encard ≤ 2,
  { intros x y,
    rw [(({x,y} : set α).to_finite.encard_eq), ←nat.cast_two,
   nat.cast_le],
    exact (ncard_insert_le _ _).trans (by simp) },
  haveI : M.finite := ⟨hfin⟩,
  refine ⟨λ h, ⟨λ e he f hf, (h _).mpr ⟨lem _ _,_⟩,_⟩, λ h I, _⟩,

  { rintro x ((rfl : x = e)| (rfl : x = f)); assumption  },
  { rw [rk],
    rw [←one_add_one_eq_two, nat.add_one_le_iff, one_lt_ncard_iff hfin]
   at hk,
    obtain ⟨a, b, ha, hb, hne⟩ := hk,
    have hss : {a,b} ⊆ M.E, by {rintro x ((rfl : x = a) | (rfl : x = b));
   assumption},
    have hlb := M.r_mono hss,
    rw [indep.r ((h _).mpr ⟨_, hss⟩), ncard_pair hne] at hlb,
    { refine hlb.antisymm' _,
      obtain ⟨B, hB⟩ := M.exists_base,
      rw [←rk, ←hB.card],
      have h' := ((h B).mp hB.indep).1,
      rw [←nat.cast_two, encard_le_coe_iff] at h',
      exact h'.2 },
    apply lem },
  rw [←nat.cast_two, encard_le_coe_iff],
  refine ⟨λ h', ⟨⟨h'.finite, _⟩, h'.subset_ground⟩, _⟩,
  { rw [←h'.r, ←h.2], exact r_le_rk _ _ },
  rintro ⟨⟨hfin, hcard⟩, hss⟩,
  rw [le_iff_eq_or_lt, nat.lt_iff_add_one_le, ncard_eq_two, ←
   one_add_one_eq_two,
    nat.add_le_add_iff_right, ncard_le_one_iff_eq hfin] at hcard,
  obtain (⟨x,y,-, rfl⟩ | rfl | ⟨e, rfl⟩ ) := hcard,
  { exact h.1 _ (hss (by simp)) _ (hss (by simp)) },
  { simp },
  convert h.1 e (hss (by simp)) e (hss (by simp)),
  simp,
end

end unif
```

```
end matroid
```

A.4 Representation Definitions & Theorem

```
/-
Copyright (c) 2023 Alena Gusakov. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Alena Gusakov
-/
import analysis.inner_product_space.gram_schmidt_ortho
import data.zmod.basic data.finsupp.fintype
import linear_algebra.linear_independent
import .constructions_mwe

universe u
variables {α γ : Type} {β 𝔽 : Type*} {M : matroid α} {I B : set α} {x : α
   }
variables {W W' : Type*} [field 𝔽] [add_comm_group W] [module 𝔽 W]
   [add_comm_group W'] [module 𝔽 W']

open function set submodule finite_dimensional

lemma set.injective_iff_forall_inj_on_pair {f : α → β} : injective f ↔ ∀
    a b, inj_on f {a, b} :=
⟨λ h a b, h.inj_on _, λ h a b hab,
  h _ _ (mem_insert _ _) (mem_insert_of_mem _ $ mem_singleton _) hab⟩

noncomputable theory


open_locale classical

namespace matroid

structure rep (𝔽 W : Type*) [field 𝔽] [add_comm_group W] [module 𝔽 W] (M :
    matroid α) :=
(to_fun : α → W)
(valid' : ∀ (I ⊆ M.E), linear_independent 𝔽 (to_fun ∘ coe : I → W) ↔
   M.indep I)
(support : ∀ (e : α), e ∉ M.E → to_fun e = 0)

instance fun_like {𝔽 W : Type*} [field 𝔽] [add_comm_group W] [module 𝔽 W]
   {M : matroid α } :
  fun_like (rep 𝔽 W M) α (λ _, W) :=
```

```
{ coe := λ φ e, φ.to_fun e,
  coe_injective' := λ f g h, by cases f; cases g; congr' }

instance : has_coe_to_fun (rep 𝔽 W M) (λ _, α → W) :=
   fun_like.has_coe_to_fun

lemma rep.valid (φ : rep 𝔽 W M) {I : set α} : linear_independent 𝔽 (λ e :
    I, φ e) ↔ M.indep I :=
begin
  refine (em (I ⊆ M.E)).elim (φ.valid' _) (fun hIE, _),
  obtain ⟨e, heI, heE⟩ := not_subset.1 hIE,
  exact iff_of_false (fun hli, hli.ne_zero ⟨e, heI⟩ (φ.support _ heE))
    (fun hI, hIE hI.subset_ground),
end

def is_representable (𝔽 : Type*) [field 𝔽] (M : matroid α) : Prop :=
  ∃ (B : set α) (hB : M.base B), nonempty (rep 𝔽 (B →₀ 𝔽) M)

namespace rep

section matroid_lemmas

lemma contract_circuit_of_insert_circuit (e : α) (C : set α) (he :
   M.nonloop e) (heC : e ∉ C)
  (hMCe : M.circuit (insert e C)) : (M / e).circuit C :=
begin
  simp_rw [circuit_iff_forall_ssubset, contract_elem,
   he.indep.contract_dep_iff, union_singleton,
    he.indep.contract_indep_iff] at *,
  refine ⟨⟨disjoint_singleton_right.2 heC, hMCe.1⟩, λ I hI,
    ⟨disjoint_singleton_right.2 (not_mem_subset (subset_of_ssubset hI)
   heC), _⟩⟩,
  have h8 : insert e I ⊂ insert e C,
    obtain ⟨a, ⟨haI, haIC⟩⟩ := ssubset_iff_insert.1 hI,
    have ha : ¬(a = e ∨ a ∈ I),
    { push_neg,
      refine ⟨ne_of_mem_of_not_mem (mem_of_mem_of_subset (mem_insert _ I)
   haIC) heC, haI⟩ },
    apply ssubset_iff_insert.2 ⟨a, ⟨mem_insert_iff.1.mt ha, _⟩⟩,
    rw insert_comm,
    apply insert_subset_insert haIC,
  rw union_singleton,
```

```
  apply hMCe.2 _ h8,
end

lemma coindep.base_of_basis_del {X : set α} (hX : M.coindep X) (hB :
   M.basis B (M.E \ X)) :
  M.base B :=
begin
  obtain ⟨B',hB'⟩ := hX.exists_disjoint_base,
  apply hB'.1.base_of_basis_supset (subset_diff.2 ⟨hB'.1.subset_ground,
   disjoint.symm hB'.2⟩) hB,
end

lemma series_pair_mem_circuit (x y : α) (C : set α) (hMC : M.circuit C)
  (hMxy : M.cocircuit {x, y}) : x ∈ C ↔ y ∈ C :=
begin
  suffices h : ∀ (M' : matroid α) {x' y' C'},
    M'.cocircuit C' → M'.circuit {x',y'} → x' ∈ C' → y' ∈ C',
  { rw [← dual_circuit_iff_cocircuit] at hMxy,
    rw [ ←dual_dual M, dual_circuit_iff_cocircuit] at hMC,
    exact ⟨h M* hMC hMxy, h M* hMC (by rwa [pair_comm])⟩ },
  clear hMC C hMxy x y M,
  refine fun M e f C hC hef heC, by_contra (fun hfC, _),
  obtain (rfl | hne) := eq_or_ne e f, exact hfC heC,
  rw [← compl_hyperplane_iff_cocircuit] at hC,
  have hss : {e,f} \ {e} ⊆ M.E \ C,
  { simp only [insert_diff_of_mem, mem_singleton,
   diff_singleton_subset_iff, singleton_subset_iff,
      mem_insert_iff, mem_diff],
    exact or.inr ⟨hef.subset_ground (or.inr rfl), hfC⟩ },

  have hcon := (hef.subset_cl_diff_singleton e).trans (M.cl_subset hss)
   (or.inl rfl),
  rw [hC.flat.cl] at hcon,
  exact hcon.2 heC,
end

end matroid_lemmas

section unif_lemmas

lemma unif_simple (a b : ℕ) (ha : 2 ≤ a) : (unif a b).simple :=
begin
```

```
  rintro e - f -,
  simp only [unif_indep_iff', nat.cast_bit0, enat.coe_one],
  have hfin : ({e,f} : set (fin b)).finite := ((finite_singleton
   _).insert _),
  rw [encard_le_coe_iff, and_iff_right hfin],
  refine le_trans _ ha,
  obtain (rfl | hne) := eq_or_ne e f, simp,
  rw [ncard_pair hne],
end

lemma delete_elem_unif (k n : ℕ) (e : fin (n + 1)) : nonempty (unif k (n +
    1) \ e ≃i unif k n) :=
begin
  rw [iso_unif_iff, delete_elem, eq_iff_indep_iff_indep_forall,
   delete_ground, unif_ground_eq,
    encard_eq_coe_iff, ncard_diff (singleton_subset_iff.2 (mem_univ e)),
   ncard_singleton,
    ncard_univ, nat.card_eq_fintype_card, fintype.card_fin,
   nat.add_succ_sub_one, add_zero],
  refine ⟨⟨rfl, λ I hI, _⟩, ⟨finite.diff (@univ (fin (n + 1))).to_finite
   {e}, rfl⟩⟩,
  simp only [← compl_eq_univ_diff, delete_indep_iff, unif_indep_iff',
   disjoint_singleton_right,
    set.unif_on_indep_iff, subset_compl_singleton_iff, encard_le_coe_iff,
   and_assoc],
end

lemma contract_elem_unif (k n : ℕ) (e : fin (n + 1)) :
  nonempty (unif (k + 1) (n + 1) / e ≃i unif k n) :=
begin
  rw [iso_unif_iff, contract_elem, eq_iff_indep_iff_indep_forall,
   contract_ground, unif_ground_eq,
    encard_eq_coe_iff, ncard_diff (singleton_subset_iff.2 (mem_univ e)),
   ncard_singleton,
    ncard_univ, nat.card_eq_fintype_card, fintype.card_fin,
   nat.add_succ_sub_one, add_zero],
  refine ⟨⟨rfl, λ I hI, _⟩, ⟨finite.diff (@univ (fin (n + 1))).to_finite
   {e}, rfl⟩⟩,
  simp only [← compl_eq_univ_diff],
  rw [indep.contract_indep_iff, unif_indep_iff', disjoint_singleton_right,
   set.unif_on_indep_iff,
    subset_compl_singleton_iff, encard_le_coe_iff, union_singleton,
```

```
    and_comm, ← and_assoc],
  refine ⟨λ h, ⟨_, h.2⟩, λ h, ⟨_, h.2⟩⟩,
  { refine ⟨h.1.1.subset (subset_insert _ _), _⟩,
    rw [← add_le_add_iff_right 1, ← ncard_insert_of_not_mem h.2],
    apply h.1.2 },
  { refine ⟨h.1.1.insert _, _⟩,
    rw [ncard_insert_of_not_mem h.2, add_le_add_iff_right],
    apply h.1.2 },
  simp only [unif_indep_iff, ncard_singleton, le_add_iff_nonneg_left,
   zero_le'],
end

lemma unif_iso_minor {n m k : ℕ} (hjk : m ≤ n) : unif k m ≤i unif k n :=
begin
  set D : set (fin n) := (range (fin.cast_le hjk))ᶜ with hD,

  have hecard : (range (fin.cast_le hjk)).encard = m,
  { rw [←image_univ,  encard_image_of_injective],
    { rw [encard_eq_coe_iff, ncard_univ, nat.card_eq_fintype_card,
        and_iff_left (fintype.card_fin _)],
      exact univ.to_finite },
    exact rel_embedding.injective (fin.cast_le hjk) },

  refine ⟨(unif k n) \  D, delete_minor _ _, ⟨iso.symm (nonempty.some _)⟩⟩,
  rw [iso_unif_iff, delete_ground, unif_ground_eq, ← compl_eq_univ_diff,
   hD, compl_compl,
    and_iff_left hecard, eq_iff_indep_iff_indep_forall],
  simp [restrict_ground_eq', encard_le_coe_iff, and_assoc],
end

end unif_lemmas

section linear_independent_lemmas

lemma linear_independent.map'' {ι : Type*} {v : ι → W} (hv :
   linear_independent 𝔽 v) (f : W →ₗ[𝔽] W')
   (hfv : linear_independent 𝔽 (f ∘ v)) : disjoint (span 𝔽 (range v))
   f.ker :=
begin
  rw [disjoint_iff_inf_le, ← set.image_univ,
   finsupp.span_image_eq_map_total,
    map_inf_eq_map_inf_comap, map_le_iff_le_comap, comap_bot,
```

```
   finsupp.supported_univ, top_inf_eq],
  unfold linear_independent at hv hfv,
  rw [hv, le_bot_iff],
  haveI : inhabited W := ⟨0⟩,
  rw [finsupp.total_comp, @finsupp.lmap_domain_total _ _ 𝔽 _ _ _ _ _ _
   _ _ _ f,
    linear_map.ker_comp (finsupp.total ι W 𝔽 v) f] at hfv,
  rw ← hfv,
  exact λ _, rfl,
end

/-- If `f` is an injective linear map, then the family `f ∘ v` is linearly
   independent
if and only if the family `v` is linearly independent and the kernel of `
   f` is disjoint from
the span of `v`. -/
protected lemma linear_map.linear_independent_iff {ι : Type*} {v : ι → W}
   (f : W →ₗ[𝔽] W') :
  linear_independent 𝔽 (f ∘ v) ↔ linear_independent 𝔽 v ∧ disjoint
   (f.ker) (span 𝔽 (range v)) :=
⟨λ h, ⟨@linear_independent.of_comp _ _ _ W' _ _ _
  (@add_comm_group.to_add_comm_monoid W' _inst_4) _ _inst_5 f h,
  disjoint.comm.1 (linear_independent.map'' (@linear_independent.of_comp _
   _ _ W' _ _ _
  (@add_comm_group.to_add_comm_monoid W' _inst_4) _ _inst_5 f h) _ h)⟩,
  λ h, linear_independent.map h.1 (disjoint.comm.1 h.2)⟩

lemma linear_independent.union' {s t : set W}
  (hs : linear_independent 𝔽 (λ x, x : s → W)) (ht : linear_independent 𝔽
    (λ x, x : t → W))
  (hst : disjoint (span 𝔽 s) (span 𝔽 t)) (hst2 : linear_independent 𝔽 (λ
   x, x : (s ∪ t) → W))
    : disjoint s t :=
begin
  rw disjoint_def at hst,
  rw [set.disjoint_iff, subset_empty_iff, eq_empty_iff_forall_not_mem],
  intros x,
  by_contra,
  -- for some reason, it doesn't let me specialize directly here.
  have h20 := mem_of_subset_of_mem (subset_span) ((mem_inter_iff _ _ _).1
   h).1,
  have h21 := mem_of_subset_of_mem (subset_span) ((mem_inter_iff _ _ _).1
```

```
   h).2,
  specialize hst x h20 h21,
  apply @linear_independent.ne_zero _ 𝔽 W ((λ (x : (s ∪ t)), x)) _ _ _ _
    ⟨x, (mem_of_subset_of_mem (inter_subset_union s t) h)⟩ hst2,
  simp only [← hst, subtype.coe_mk],
end

lemma linear_independent.union'' {s t : set W}
  (hs : linear_independent 𝔽 (λ x, x : s → W)) (ht : linear_independent 𝔽
    (λ x, x : t → W))
  (hst : disjoint s t) (hst2 : linear_independent 𝔽 (λ x, x : (s ∪ t) →
   W))
    :  disjoint (span 𝔽 s) (span 𝔽 t) :=
begin
  convert hst2.disjoint_span_image (show disjoint (coe ⁻¹' s) (coe ⁻¹' t),
   from _),
  { rw [eq_comm, image_preimage_eq_iff, subtype.range_coe], apply
   subset_union_left },
  { rw [eq_comm, image_preimage_eq_iff, subtype.range_coe], apply
   subset_union_right },
  rw [set.disjoint_iff, subset_empty_iff] at ⊢ hst,
  rw [← preimage_inter, hst, preimage_empty],
end

end linear_independent_lemmas

section rep_lemmas

lemma inj_on_of_indep (φ : rep 𝔽 W M) (hI : M.indep I) : inj_on φ I :=
inj_on_iff_injective.2 ((φ.valid' I hI.subset_ground).2 hI).injective

@[simp] lemma to_fun_eq_coe (φ : rep 𝔽 W M) : φ.to_fun = (φ : α → W)  :=
    by { ext, refl }

lemma support' {φ : rep 𝔽 W M} {e : α} (he : e ∉ M.E) : φ e = 0 :=
by { rw ← to_fun_eq_coe, apply φ.support _ he }

def rep_of_congr {M M' : matroid α} (φ : rep 𝔽 W M) (h : M = M') : rep 𝔽
   W M' :=
{ to_fun := φ.to_fun,
  valid' := λ I hI, by { rw ← (eq_iff_indep_iff_indep_forall.1 h).1 at hI,
    rw ← (eq_iff_indep_iff_indep_forall.1 h).2, apply φ.valid' I hI,
```

```
    apply hI },
  support := λ e he, by { rw ← (eq_iff_indep_iff_indep_forall.1 h).1 at
    he, apply φ.support e he } }

def rep_of_iso (M : matroid α) (M' : matroid γ) (ψ : M' ≃i M) (v : rep 𝔽
    W M) : rep 𝔽 W M' :=
{ to_fun := function.extend coe (fun (x : M'.E), v (ψ x)) 0,
  valid' := λ I hI,
    begin
      set eI : I → ψ.image I := λ x, ⟨ψ ⟨x,hI x.2⟩, ⟨_,mem_image_of_mem
    _ (by simp), rfl⟩⟩ with heI,
      have hbij : bijective eI,
      { refine ⟨fun x y hxy, _, fun x, _⟩,
        { rwa [heI, subtype.mk_eq_mk, subtype.coe_inj,
    (equiv_like.injective ψ).eq_iff,
            subtype.mk_eq_mk, subtype.coe_inj] at hxy },
        obtain ⟨_, ⟨_, ⟨z,hz,rfl⟩, rfl⟩⟩ := x,
        exact ⟨⟨z,hz⟩, by simp⟩ },
      rw [ψ.on_indep hI, ← v.valid ],
      refine linear_independent_equiv' (equiv.of_bijective _ hbij) _,
      ext,
      simp only [comp_app, equiv.of_bijective_apply, subtype.coe_mk],
      exact ((@subtype.coe_injective _ M'.E).extend_apply (λ x, v (ψ x))
    0 (inclusion hI x)).symm,
    end,
  support :=
    begin
      rintro e he,
      rw [extend_apply', pi.zero_apply],
      rintro ⟨a,rfl⟩,
      exact he a.2,
    end }

lemma ne_zero_of_nonloop (φ : rep 𝔽 W M) (hx : M.nonloop x) : φ x ≠ 0 :=
((φ.valid' {x} (indep_singleton.2 hx).subset_ground).2 hx.indep).ne_zero
(⟨x, mem_singleton _⟩ : ({x} : set α))

lemma ne_zero_of_loopless (φ : rep 𝔽 W M) (hl : loopless M) (x : α) (hx :
    x ∈ M.E) : φ x ≠ 0 :=
 φ.ne_zero_of_nonloop (hl x hx)

lemma inj_on_ground_of_simple (φ : rep 𝔽 W M) (hs : simple M) : inj_on φ
```

```
   M.E :=
λ a ha b hb,
begin
  apply φ.inj_on_of_indep (hs a ha b hb),
  simp only [mem_insert_iff, eq_self_iff_true, true_or],
  simp only [mem_insert_iff, eq_self_iff_true, mem_singleton, or_true],
end

lemma subset_nonzero_of_simple (φ : rep 𝔽 W M) (hs : simple M) :
  φ '' M.E ⊆ span 𝔽 (φ '' M.E) \ {0} :=
begin
  refine subset_diff.2 ⟨subset_span, disjoint_left.2 _⟩,
  rintro x ⟨y, ⟨hy1, rfl⟩⟩,
  apply ne_zero_of_loopless _ hs.loopless _ hy1,
end

lemma of_basis (φ : rep 𝔽 W M) {X I : set α} (hI : M.basis I X) {e : α}
    (he : e ∈ X):
  φ e ∈ span 𝔽 (φ '' I) :=
begin
  by_cases e ∈ I,
  { apply subset_span (mem_image_of_mem _ h) },
  have h2 : ¬ linear_independent 𝔽 (λ x : insert e I, φ x) := (φ.valid'
    (insert e I)
  (insert_subset.2 ⟨(mem_of_mem_of_subset he hI.subset_ground),
   hI.subset_ground_left⟩)).not.2
  (dep_iff.1 (hI.insert_dep (mem_diff_of_mem he h))).1,
  contrapose! h2,
  apply (linear_independent_insert' h).2 ⟨(φ.valid' I
   hI.subset_ground_left).2 hI.indep, h2⟩,
end

lemma span_basis (φ : rep 𝔽 W M) {X I : set α} (hI : M.basis I X) :
  span 𝔽 (φ '' I) = span 𝔽 (φ '' X) :=
begin
  refine (span_mono $ image_subset _ (basis.subset hI)).antisymm
    (span_le.2 _),
  rintros _ ⟨y, ⟨hy1, rfl⟩⟩,
  apply of_basis φ hI hy1,
end

lemma span_base (φ : rep 𝔽 W M) (hB : M.base B) : span 𝔽 (φ '' B) = span 𝔽
```

```
    (φ '' M.E) :=
  by { rw [span_basis φ (base.basis_ground hB)] }

@[simp] lemma mem_span_rep_range (φ : rep 𝔽 W M) : ∀ (x : α), φ x ∈
    (span 𝔽 (range ⇑φ)) :=
  λ x, by { apply mem_of_subset_of_mem (@subset_span 𝔽 _ _ _ _ (range ⇑φ
    )) (mem_range_self x) }

@[simp] lemma mem_span_rep (φ : rep 𝔽 W M) : ∀ (x : α) , φ x ∈ (span 𝔽
    (φ '' M.E)) :=
  λ x, by { by_cases x ∈ M.E,
apply mem_of_subset_of_mem (@subset_span 𝔽 _ _ _ _ (φ '' M.E))
    (mem_image_of_mem φ h),
simp only [support' h, submodule.zero_mem] }

@[simp]
lemma span_range_eq_span_image (φ : rep 𝔽 W M) : span 𝔽 (φ '' M.E) = span
    𝔽 (range ⇑φ) :=
span_eq_span (λ x ⟨y, ⟨hx1, hx2⟩⟩, by {rw ← hx2, apply mem_span_rep_range
    φ y})
  (λ x ⟨y, hx⟩, by {rw ← hx, apply mem_span_rep φ y })

lemma span_range_base (φ : rep 𝔽 W M) (hB: M.base B) :
  span 𝔽 (range (λ (e : ↥B), φ ↑e)) = span 𝔽 (range φ)  :=
begin
  rw [← span_range_eq_span_image, ← φ.span_base hB],
  have h2 : range (λ (e : ↥B), φ ↑e) = (⇑φ '' B),
    ext;
    refine ⟨λ ⟨y, hy⟩, by { simp only at hy, rw ← hy, apply
   mem_image_of_mem φ y.2}, λ hx, _⟩,
    obtain ⟨y, ⟨hy1, rfl⟩⟩ := hx,
    simp only [mem_range, set_coe.exists, subtype.coe_mk, exists_prop],
    refine ⟨y, ⟨hy1, rfl⟩⟩,
  rw h2,
end

lemma fund_circuit_inter_eq_diff_of_not_mem (e : α) (he : e ∈ M.cl I) (h2
    : e ∉ I) :
  (M.fund_circuit e I ∩ I) = (M.fund_circuit e I \ {e}) :=
begin
  apply eq_of_subset_of_subset,
  rw [diff_eq, compl_eq_univ_diff],
```

```
  apply inter_subset_inter (subset.refl _) (subset_diff_singleton
   (subset_univ I) h2),
  apply subset_inter (diff_subset _ _),
  apply (@insert_subset_insert_iff _ _ ((M.fund_circuit e I) \ {e}) I
    (not_mem_diff_singleton e (M.fund_circuit e I))).1,
  rw [insert_diff_singleton, insert_eq_of_mem (mem_fund_circuit _ _ _)],
  apply fund_circuit_subset_insert he,
end

-- modify to disjoint union of circuits for iff?
lemma rep.circuit (φ : rep 𝔽 W M) {C : set α} (hMC : M.circuit C) :
  ∃ f : α →₀ 𝔽, (f.support : set α) = C ∧ finsupp.total α W 𝔽 φ f = 0 ∧
    f ≠ 0 :=
begin
  obtain ⟨f, ⟨hfssup, ⟨hftot, hfne0⟩⟩⟩ :=
    linear_dependent_comp_subtype'.1 (φ.valid.1.mt (not_indep_iff.2
   hMC.dep)),
  refine ⟨f, ⟨_, ⟨hftot, hfne0⟩⟩⟩,
  apply subset.antisymm_iff.2 ⟨hfssup, λ x hx, _⟩,
  by_contra,
  apply φ.valid.2.mt
    (linear_dependent_comp_subtype'.2 ⟨f, ⟨subset_diff_singleton hfssup h,
   ⟨hftot, hfne0⟩⟩⟩),
  apply hMC.diff_singleton_indep hx,
end

lemma mem_span_set_rep_of_not_mem (φ : rep 𝔽 W M) {I : set α} (hI :
   M.indep I)
(e : α) (he : e ∈ M.cl I) (he2 : e ∉ I) :
 ∃ c : W →₀ 𝔽, (c.support : set W) = φ '' (M.fund_circuit e I \ {e}) ∧
  c.sum (λ mi r, r • mi) = φ e :=
begin
  obtain ⟨c, ⟨hc1, hc2⟩⟩ := mem_span_set.1 (of_basis φ
   (circuit.diff_singleton_basis
    (indep.fund_circuit_circuit hI ((mem_diff e).2 ⟨he, he2⟩)))
   (M.mem_fund_circuit e I))
    (M.mem_fund_circuit e I)),
  refine ⟨c, ⟨subset.antisymm_iff.2 ⟨hc1, λ x hx, _⟩, hc2⟩⟩,
  obtain ⟨y, ⟨⟨hy1, hy2⟩, rfl⟩⟩ := hx,
  by_contra,
  have h5 : ∃ (c : W →₀ 𝔽), ↑(c.support) ⊆ ⇑φ '' (M.fund_circuit e I \
   {e}) \ {φ y} ∧
```

```
  c.sum (λ (mi : W) (r : 𝔽), r • mi) = φ e,
refine ⟨c, ⟨subset_diff_singleton hc1 h, hc2⟩⟩,
have h8 : e ∈ ((M.fund_circuit e I) \ {y}),
{ simp only [mem_diff, mem_singleton_iff],
  refine ⟨(M.mem_fund_circuit e I), ne.symm hy2⟩ },
have h7 := (linear_independent_iff_not_mem_span.1 ((φ.valid'
 (M.fund_circuit e I \ {y})
  (subset.trans (diff_subset _ _) (fund_circuit_subset_ground he))).2
  (circuit.diff_singleton_indep
  (indep.fund_circuit_circuit hI ((mem_diff e).2 ⟨he, he2⟩)) hy1))) ⟨e,
 h8⟩,
simp only [subtype.coe_mk, to_fun_eq_coe] at h7,
rw [set.image_eta] at h7,
have h9 : ((λ (a : ↥(M.fund_circuit e I \ {y})), φ ↑a) '' (univ \ {⟨e,
 h8⟩})) =
  (⇑φ '' (M.fund_circuit e I \ {e}) \ {φ y}),
{ ext;
  refine ⟨λ h, _, λ h20, _⟩,
  { simp only [mem_image, mem_diff, mem_univ, mem_singleton_iff,
 true_and, set_coe.exists,
      subtype.mk_eq_mk, subtype.coe_mk, exists_prop] at h,
    obtain ⟨a, ⟨⟨ha1, ha2⟩, ⟨ha3, rfl⟩⟩⟩ := h,
    simp only [mem_diff, mem_image, mem_singleton_iff],
    use ⟨a, ⟨⟨ha1, ha3⟩, rfl⟩⟩,
    have h11 : (insert y {a}) ⊂ M.fund_circuit e I,
    rw ssubset_iff_subset_diff_singleton,
    refine ⟨e, ⟨(M.mem_fund_circuit e I), λ x hx, _⟩⟩,
    obtain ⟨rfl, rfl⟩ := hx,
    rw mem_diff_singleton,
    simp only [mem_singleton_iff] at hy2,
    refine ⟨hy1, hy2⟩,
    rw mem_diff_singleton,
    simp only [mem_singleton_iff] at hx,
    rw hx,
    refine ⟨ha1, ha3⟩,
    have h10 := inj_on_of_indep φ
      (circuit.ssubset_indep (indep.fund_circuit_circuit hI ((mem_diff
 e).2 ⟨he, he2⟩)) h11),
    apply (inj_on.ne_iff h10 _ _).2 ha2,
    simp only [mem_insert_iff, mem_singleton, or_true],
    simp only [mem_insert_iff, eq_self_iff_true, true_or]},
 { obtain ⟨⟨a, ⟨⟨ha1, ha2⟩, rfl⟩⟩, ha3⟩ := h20,
```

```
      use a,
      apply mem_diff_singleton.2 ⟨ha1, _⟩,
      simp only [mem_singleton_iff] at ha3,
      by_contra,
      rw h at ha3,
      apply ha3,
      refl,
      refine ⟨(mem_diff _).2 ⟨mem_univ _, mem_singleton_iff.1.mt
   (subtype.mk_eq_mk.1.mt ha2)⟩, _⟩,
      simp only [subtype.coe_mk]} },
  rw h9 at h7,
  apply h7,
  exact mem_span_set.2 h5,
end

end rep_lemmas

section standard_rep_lemmas

/-- The representation for `M` whose rows are indexed by a base `B` -/
def standard_rep (φ' : rep 𝔽 W M) {B : set α} (hB : M.base B) :
  rep 𝔽 (B →₀ 𝔽) M :=
{ to_fun := λ e : α, ((valid φ').2 hB.indep).repr ⟨φ' e, by
    { have h4 := φ'.mem_span_rep_range, rw ← span_range_base φ' hB at h4,
   exact h4 e}⟩,
  valid' := by
  { intros I hI,
    rw [← @valid _ _ _ _ _ _ _ _ φ' I,
      linear_map.linear_independent_iff ((valid φ').2 hB.indep).repr,
      ←(submodule.subtype (span 𝔽 (range (λ (e : B), φ' ↑
   e)))).linear_independent_iff,
        submodule.coe_subtype, and_iff_left],
    { refl },
    { simp only [linear_independent.repr_ker, disjoint_bot_left] },
    simp only [ker_subtype] },
  support := by
  { intros e he, simp_rw [support' he], convert _root_.map_zero _} }

@[simp]
lemma id_matrix_of_base (φ : rep 𝔽 W M) {B : set α} (e : B) (hB : M.base
   B) :
  standard_rep φ hB e e = 1 :=
```

```
begin
  rw ← to_fun_eq_coe,
  simp [standard_rep],
  rw [((valid φ).2 hB.indep).repr_eq_single e ⟨φ e, by
    { have h4 := φ.mem_span_rep_range, rw ← span_range_base φ hB at h4,
   exact h4 e}⟩ rfl],
  simp only [finsupp.single_eq_same],
end

lemma id_matrix_of_base' (φ : rep 𝔽 W M) {B : set α} (e f : B) (hB :
    M.base B) (hne : e ≠ f) :
  standard_rep φ hB e f = 0 :=
begin
  rw ← to_fun_eq_coe,
  simp [standard_rep],
  rw [(φ.valid.2 hB.indep).repr_eq_single e ⟨φ e, by
    { have h4 := φ.mem_span_rep_range, rw ← span_range_base φ hB at h4,
   exact h4 e}⟩ rfl],
  apply finsupp.single_eq_of_ne hne,
end

lemma standard_rep_base_eq {M' : matroid α} (φ : rep 𝔽 W M) (φ' : rep 𝔽 W'
    M') {B : set α}
(hB : M.base B) (hB' : M'.base B) (e : B) : standard_rep φ hB e =
    standard_rep φ' hB' e :=
begin
  ext;
  by_cases e = a,
  simp_rw [h, id_matrix_of_base],
  simp_rw [id_matrix_of_base' φ e a hB h, id_matrix_of_base' φ' e a hB' h],
end

lemma standard_rep_eq_of_congr {M M' : matroid α} (φ : rep 𝔽 W M) (h : M =
     M') {B : set α}
  (hMB : M.base B) (hMB' : M'.base B) :
  ((standard_rep φ hMB) : α → B →₀ 𝔽) = (standard_rep (rep_of_congr φ
   h) hMB' :  α → B →₀ 𝔽) := rfl

/-- A representation over *any* module certifies representability-/
lemma is_representable_of_rep {W : Type*} [add_comm_group W] [module 𝔽 W]
    (φ : rep 𝔽 W M) :
    is_representable 𝔽 M :=
```

```
  begin
    obtain ⟨B, hB⟩ := M.exists_base,
    exact ⟨B, hB, ⟨standard_rep φ hB⟩⟩,
  end

end standard_rep_lemmas

section matroid_of_module_fun

def matroid_of_module_fun (𝔽 W : Type*) {ι : Type*} [field 𝔽]
   [add_comm_group W] [module 𝔽 W]
  [finite_dimensional 𝔽 W] (v : ι → W) (ground : set ι) :
  matroid ι := matroid_of_indep_of_bdd' ground
  (λ (I : set ι), (linear_independent 𝔽 (λ x : I, v x)) ∧ I ⊆ ground)
  begin
    rw [linear_independent_image (inj_on_empty _), image_empty],
    refine ⟨linear_independent_empty 𝔽 W, empty_subset ground⟩,
  end
  begin
    intros I J hJ hIJ,
    have hIJ3 := linear_independent.injective hJ.1,
    rw [← set.restrict, ← inj_on_iff_injective] at hIJ3,
    rw linear_independent_image hIJ3 at hJ,
    rw linear_independent_image (inj_on.mono hIJ hIJ3),
    refine ⟨linear_independent.mono (image_subset v hIJ) hJ.1,
   subset_trans hIJ hJ.2⟩,
  end
  begin
    intros I J hI hJ hIJ,
    have h3 : ∃ x ∈ J, v x ∉ span 𝔽 (v '' I),
    { have hJ2 := linear_independent.injective hJ.1,
      rw [← set.restrict, ← inj_on_iff_injective] at hJ2,
      rw linear_independent_image hJ2 at hJ,
      have hI2 := linear_independent.injective hI.1,
      rw [← set.restrict, ← inj_on_iff_injective] at hI2,
      rw linear_independent_image hI2 at hI,
      haveI := finite.fintype (_root_.linear_independent.finite hI.1),
      haveI := finite.fintype (_root_.linear_independent.finite hJ.1),
      by_contra,
      push_neg at h,
      have h8 : ((v '' J).to_finite.to_finset) = (v '' J).to_finset,
        ext,
```

```
      simp only [finite.mem_to_finset, mem_to_finset],
    have h9 : ((v '' I).to_finite.to_finset) = (v '' I).to_finset,
      ext,
      simp only [finite.mem_to_finset, mem_to_finset],
    have h5 : (v '' I).ncard < (v '' J).ncard,
    { rwa [ncard_image_of_inj_on hJ2, ncard_image_of_inj_on hI2] },
    apply not_le_of_lt h5,
    rw [ncard_eq_to_finset_card, ncard_eq_to_finset_card, h8, h9,
    ← finrank_span_set_eq_card (v '' I) hI.1, ←
  finrank_span_set_eq_card (v '' J) hJ.1],
    have h2 := (@span_le 𝔽 W _ _ _ (v '' J) (span 𝔽 (v '' I))).2 (λ j hj,
  _),
    swap,
    { obtain ⟨x, ⟨hx, rfl⟩⟩ := hj,
      apply h x hx },
    apply submodule.finrank_le_finrank_of_le h2 },
   obtain ⟨x, ⟨hx1, hx2⟩⟩ := h3,
   refine ⟨x, ⟨hx1, ⟨(mem_image_of_mem v).mt (not_mem_subset
  (subset_span) hx2), _⟩⟩⟩,
   refine ⟨(linear_independent_insert' ((mem_image_of_mem v).mt
    (not_mem_subset (subset_span) hx2))).2 ⟨hI.1, hx2⟩, insert_subset.2
      ⟨(mem_of_subset_of_mem hJ.2 hx1), hI.2⟩⟩,
 end
 begin
   refine ⟨finite_dimensional.finrank 𝔽 W, λ I hI, _⟩,
   have hI2 := linear_independent.injective hI.1,
    rw [← set.restrict, ← inj_on_iff_injective] at hI2,
    rw linear_independent_image hI2 at hI,
   haveI := finite.fintype (_root_.linear_independent.finite hI.1),
   rw ← linear_independent_image hI2 at hI,
   haveI := ((v '' I).to_finite.of_finite_image hI2).fintype,

   rw [ncard, nat.card_eq_fintype_card],
   refine ⟨to_finite I, fintype_card_le_finrank_of_linear_independent
  hI.1⟩,

 end
 (by { tauto })

lemma matroid_of_module_fun.ground (𝔽 W : Type*) {ι : Type*} [field 𝔽]
   [add_comm_group W] [module 𝔽 W]
 [finite_dimensional 𝔽 W] (v : ι → W) (ground : set ι) :
```

```
    (matroid_of_module_fun 𝔽 W v ground).E = ground :=
begin
  rw [matroid_of_module_fun, matroid_of_indep_of_bdd',
   matroid_of_indep_of_bdd,
    matroid_of_indep, matroid_of_base, ← ground_eq_E],
end

lemma matroid_of_module_fun_congr (𝔽 W : Type*) {ι : Type*} [field 𝔽]
   [add_comm_group W] [module 𝔽 W]
  [finite_dimensional 𝔽 W] (v w : ι → W) (ground : set ι) (hvw : ∀ (e : ι
   ), e ∈ ground → v e = w e) :
  matroid_of_module_fun 𝔽 W v ground = matroid_of_module_fun 𝔽 W w ground
   :=
begin
  apply eq_of_indep_iff_indep_forall,
  simp only [matroid_of_module_fun.ground],
  intros I hI,
  simp only [matroid_of_module_fun, matroid_of_indep_of_bdd',
   matroid_of_indep_of_bdd_apply,
    λ e : I, hvw e (mem_of_mem_of_subset e.2 hI)],
end

lemma delete_matroid_of_module_fun (𝔽 W : Type*) {ι : Type*} [field 𝔽]
   [add_comm_group W] [module 𝔽 W]
  [finite_dimensional 𝔽 W] (v : ι → W) (ground : set ι) (D : set ι) :
    matroid_of_module_fun 𝔽 W v (ground \ D) = (matroid_of_module_fun 𝔽 W
   v ground) \ D :=
begin
  apply eq_of_indep_iff_indep_forall,
  simp only [delete_ground, matroid_of_module_fun.ground],
  intros I hI,
  simp only [delete_indep_iff, matroid_of_module_fun,
   matroid_of_indep_of_bdd', subset_diff,
    matroid_of_indep_of_bdd_apply, and_assoc],
end

lemma matroid_of_module_fun_rep_eq (M : matroid α) (𝔽 W : Type*) [field 𝔽
   ] [add_comm_group W]
  [module 𝔽 W] [finite_dimensional 𝔽 W] (φ : rep 𝔽 W M) :
  M = matroid_of_module_fun 𝔽 W φ M.E :=
begin
  apply eq_of_indep_iff_indep_forall _ (λ I hI, _),
```

```
  refl,
  have hset : (λ (x : ↥I), φ x) = (φ.to_fun ∘ coe),
  { ext,
    simp only [comp_app],
    refl },
  rw [matroid_of_module_fun, matroid_of_indep_of_bdd'_apply, hset, φ
   .valid'],
  refine ⟨λ h, ⟨h, hI⟩, λ h, h.1⟩,
  apply hI,
end

def rep_of_matroid_of_module_fun (𝔽 W : Type*) {ι : Type*} [field 𝔽]
    [add_comm_group W] [module 𝔽 W]
  [finite_dimensional 𝔽 W] (v : ι → W) (ground : set ι) : rep 𝔽 W
    (matroid_of_module_fun 𝔽 W v ground) :=
{ to_fun := λ x, if x ∈ ground then v x else 0,
  valid' := λ I hI, by {simp only [matroid_of_module_fun,
   matroid_of_indep_of_bdd'_apply],
    rw matroid_of_module_fun.ground at hI,
    have h2 : (λ (x : ι), if (x ∈ ground) then (v x) else 0) ∘ (coe : I →
    ι) = λ x : I, v x,
      ext;
      simp only [ite_eq_left_iff],
      contrapose,
      intros h,
      push_neg,
      apply mem_of_subset_of_mem hI x.2,
    rw h2,
    simp,
    intros h,
    apply hI },
  support := λ e he,
    begin
      simp only [ite_eq_iff],
      right,
      refine ⟨he, rfl⟩,
    end }

end matroid_of_module_fun

section binary_lemmas
```

```
/- A matroid is binary if it has a `GF(2)`-representation -/
@[reducible, inline] def matroid.is_binary (M : matroid α) :=
   M.is_representable (zmod 2)

open_locale big_operators

lemma mem_sum_basis_zmod2_of_not_mem [fintype α] [module (zmod 2) W] (φ :
   rep (zmod 2) W M)
{I : set α} (hI : M.indep I) (e : α) (he : e ∈ M.cl I) (heI : e ∉ I) :
  Σ i in (M.fund_circuit e I \ {e}).to_finset, φ i = φ e :=
begin
  have h3 := subset_insert e (M.fund_circuit e I),
  obtain ⟨c, ⟨hc1, hc2⟩⟩ := mem_span_set_rep_of_not_mem φ hI e he heI,
  rw ← hc2,
  have h4 : c.support = (φ '' (M.fund_circuit e I \ {e})).to_finset :=
    by { simp_rw [← hc1, finset.to_finset_coe] },
  have h7 : (∀ (i : W), i ∈ (⇑φ '' (M.fund_circuit e I \ {e})).to_finset
    →
    (λ (mi : W) (r : zmod 2), r • mi) i 0 = 0),
  intros x hx,
  simp only [zero_smul],
  rw [finsupp.sum_of_support_subset c (finset.subset_of_eq h4) (λ mi r, r •
    mi) h7,
    to_finset_image, to_finset_diff, finset.sum_image, finset.sum_congr],
  simp only [eq_self_iff_true],
  { intros x hx,
    simp only,
    haveI := (@add_comm_group.to_add_comm_monoid W _inst_2),
    -- for some reason i have to do this roundabout way of using
   one_smul because
    -- i can't figure out how to make my monoid instance work for it
    have hc : c (φ x) = 1,
    cases le_iff_lt_or_eq.1 (nat.le_of_lt_succ (zmod.val_lt (c (φ x))))
   with h0 h1,
    { by_contra,
      simp only [nat.lt_one_iff, zmod.val_eq_zero] at h0,
      rw ← to_finset_diff at hx,
      have hφ := finset.mem_image_of_mem φ hx,
      rw [← to_finset_image, ← h4, finsupp.mem_support_iff, ne.def] at
   hφ,
      apply hφ,
      exact h0 },
```

```
    { rw [← zmod.nat_cast_zmod_val (c (φ x)), h1, algebra_map.coe_one] },
    rw hc,
    simp only [one_smul] },
  { simp_rw [←set.to_finset_diff, mem_to_finset],
    apply inj_on_of_indep φ (circuit.diff_singleton_indep
      (indep.fund_circuit_circuit hI ((mem_diff e).2 ⟨he, heI⟩)))
   (M.mem_fund_circuit e I)) },
end

/-- If `φ` is a representation of `M` in `zmod 2`, `I` is an independent set
    of `M`, and
  `e` is a member of the closure of `I`, then `φ e` can be uniquely written
    as the sum of
  `φ i` for `i` contained in the intersection of the fundamental circuit
    of `I` and `e`, and `I` -/
lemma mem_sum_basis_zmod2 [fintype α] [module (zmod 2) W] (φ : rep (zmod
    2) W M) {I : set α}
(hI : M.indep I) (e : α) (he : e ∈ M.cl I) :
  φ e = ∑ i in (M.fund_circuit e I ∩ I).to_finset, φ i :=
begin
  by_cases e ∈ I,
  rw [hI.fund_circuit_eq_of_mem h, @to_finset_congr _ ({e}∩I) {e} _ _
    (singleton_inter_eq_of_mem h),
      to_finset_singleton, finset.sum_singleton],
  rw to_finset_congr (fund_circuit_inter_eq_diff_of_not_mem _ he h),
  apply eq.symm (mem_sum_basis_zmod2_of_not_mem φ hI e he h),
end

/-- If `M, M'` are matroids with equal ground sets with representations `φ
    M, φM'` in `zmod 2`,
  `B : set α` is a `base` of `M` and `M'`, and for all `e ∈ M.E`,
  `M.fund_circuit e B = M'.fund_circuit e B`, then `M = M'`  -/
lemma eq_of_forall_fund_circuit_eq [fintype α] {M M' : matroid α} [module
    (zmod 2) W]
[module (zmod 2) W'] (φM : rep (zmod 2) W M) (φM' : rep (zmod 2) W' M')
(hE : M.E = M'.E) (hB : M.base B) (hB' : M'.base B)
(he : ∀ e ∈ M.E, M.fund_circuit e B = M'.fund_circuit e B) :
  M = M' :=
begin
  have φM := standard_rep φM hB,
  have φM' := standard_rep φM' hB',
  apply eq_of_indep_iff_indep_forall hE,
```

```
  intros I hI,
  have hI' := hI,
  rw hE at hI',
  rw [← (standard_rep φM hB).valid' _ hI, ← (standard_rep φM' hB').valid'
   _ hI'],
  have h2 : (standard_rep φM hB).to_fun ∘ coe = λ i : I, (standard_rep φM
   hB).to_fun i,
    simp only [eq_self_iff_true],
  have h3 : (standard_rep φM' hB').to_fun ∘ coe = λ i : I, (standard_rep φ
   M' hB').to_fun i,
    simp only [eq_self_iff_true],
  rw [h2, h3],
  simp only [to_fun_eq_coe],
  simp_rw [λ (e : I), (standard_rep φM hB).mem_sum_basis_zmod2 hB.indep _
   (@base.mem_cl _ M B hB e (hI e.2)),
    λ (e : I), (standard_rep φM' hB').mem_sum_basis_zmod2 hB'.indep _
   (@base.mem_cl _ M' B hB' e (hI' e.2))],
  simp_rw λ (e : I), he e (hI e.2),
  have h6 : (λ (i : ↥I), Σ (x : α) in (M'.fund_circuit ↑i B ∩
   B).to_finset, (standard_rep φM hB) x)
    = (λ (i : ↥I), Σ (x : α) in (M'.fund_circuit ↑i B ∩ B).to_finset,
   (standard_rep φM' hB') x),
    simp only,
    have h10 :=  λ (i : ↥I), @finset.sum_congr _ _ (M'.fund_circuit i B ∩
   B).to_finset
      (M'.fund_circuit i B ∩ B).to_finset (standard_rep φM hB)
   (standard_rep φM' hB') _ rfl _,
    simp_rw  [λ (i : I), h10 i],
    intros x hx,
    rw mem_to_finset at hx,
    have h12 := standard_rep_base_eq φM φM' hB hB' ⟨x,
   (mem_of_mem_inter_right hx)⟩,
    simp only [subtype.coe_mk] at h12,
    rw h12,
  simp_rw h6,
end

-- part (iii) in the proof of theorem 6.5.4
/-- If `MI` and `MC` are two matroids with the same rank and ground set, `
   Z ⊆ J ⊆ MI.E`, and `x, y ∈ Z` are distinct, if
    - `{x, y}` is coindependent in `MI` and `MC`
    - `MI \ x = MC \ x` and `MI \ y = MC \ y`
```

```
    - `Z` is independent in `MI` and dependent in `MC`
    - `J` is independent in `MI`
    then `J = {x, y}`. -/
lemma indep_eq_doubleton_of_subset [fintype α] (MI MC : matroid α)
   [finite_rk MI] [finite_rk MC]
  (hrk : MI.rk = MC.rk) (hIC : MI.E = MC.E) (x y : α) (hxy : x ≠ y)
  (hiIC : MI.coindep {x,y} ∨ MC.coindep {x,y}) (hMx : MI \ x = MC \ x)
   (hMy : MI \ y = MC \ y)
  {Z J : set α} (hxZ : x ∈ Z) (hyZ : y ∈ Z) (hMIZ : MI.indep Z) (hMCZ : ¬
    MC.indep Z)
  (hZJ : Z ⊆ J) (hMIJ : MI.indep J) [module (zmod 2) W] [module (zmod 2)
   W']
  (φI : rep (zmod 2) W (MI / (J \ {x, y})))
  (φC : rep (zmod 2) W' (MC / (J \ {x, y}))) : J = {x, y} :=
begin
  apply subset_antisymm _ (insert_subset.2 ⟨hZJ hxZ,
   singleton_subset_iff.2 (hZJ hyZ)⟩),
  rw ← diff_eq_empty,
  by_contra,
  --have hMIxy : (MI \ {x, y}).indep (J \ {x, y}),
  rw [MI.delete_elem x, MC.delete_elem x] at hMx, --← delete_delete,
  rw [MI.delete_elem y, MC.delete_elem y] at hMy,
  have hMIxyJ := delete_indep_iff.2 ⟨hMIJ.subset (diff_subset J {x, y}),
    @disjoint_sdiff_left _ {x, y} J⟩,
  have hMIxyJ2 := hMIxyJ,
  rw [← union_singleton, ← delete_delete, hMy,
    delete_delete, union_singleton] at hMIxyJ2,
  have hNIC : (MI / (J \ {x, y})).rk = (MC / (J \ {x, y})).rk,
    { -- this is due to M and M' having the same rank
      have h2 := MI.er_contract_add_er_eq_er_union (J \ {x, y}) (MI.E \
   (J \ {x, y})),
      have h3 := MC.er_contract_add_er_eq_er_union (J \ {x, y}) (MC.E \
   (J \ {x, y})),
      rw [union_comm, union_diff_cancel (subset_trans (diff_subset _ _)
   (hMIJ.subset_ground))] at h2,
      rw [union_comm, union_diff_cancel] at h3,
      have h4 : MI.er (J \ {x, y}) = MC.er (J \ {x, y}),
      { rw [← union_singleton, ← diff_diff, ← MI.delete_er_eq', ←
   MC.delete_er_eq', hMx] },
      rw [rk_def, rk_def, ← er_eq_coe_iff, eq_comm] at hrk,
      simp only [contract_ground, coe_r_eq_er] at hrk,
      rw [hrk, ← h2, h4] at h3,
```

```
    simp only [← coe_r_eq_er, ← enat.coe_add] at h3,
    have h7 : ((MC / (J \ {x, y})).r (MC.E \ (J \ {x, y})) + MC.r (J \
 {x, y})) =
      ((MI / (J \ {x, y})).r (MI.E \ (J \ {x, y})) + MC.r (J \ {x, y})),
    { rwa [enat.coe_inj] at h3 },
    simp only [rk_def],
    rw eq_comm,
    simp only [contract_ground],
    apply nat.add_right_cancel h7,
    rw ← hIC,
    apply subset_trans (diff_subset _ _) (hMIJ.subset_ground) },
have hNIneNC : (MI / (J \ {x, y})) ≠ (MC / (J \ {x, y})),
{ simp only [ne.def, eq_iff_indep_iff_indep_forall, contract_ground,
 hIC, eq_self_iff_true,
    true_and, not_forall, exists_prop],
  refine ⟨{x, y}, ⟨_, _⟩⟩,
  { rw subset_diff,
    refine ⟨_, @disjoint_sdiff_right _ {x, y} J⟩,
    rw ← hIC,
    apply (insert_subset.2 ⟨(hMIZ.subset_ground) hxZ,
 singleton_subset_iff.2
      ((hMIZ.subset_ground) hyZ)⟩) },
  { rw [iff_def, not_and_distrib],
    left,
    push_neg,
    refine ⟨(indep.contract_indep_iff (hMIJ.subset (diff_subset J {x,
 y}))).2
      ⟨@disjoint_sdiff_right _ {x, y} J, _⟩, _⟩,
    rw union_diff_cancel (insert_subset.2 ⟨hZJ hxZ,
 singleton_subset_iff.2 (hZJ hyZ)⟩),
    apply hMIJ,
    rw [indep.contract_indep_iff (hMIxyJ2.of_delete), not_and_distrib],
    right,
    rw union_diff_cancel (insert_subset.2 ⟨hZJ hxZ,
 singleton_subset_iff.2 (hZJ hyZ)⟩),
    apply indep.subset.mt (not_imp.2 ⟨hZJ, hMCZ⟩) } },
obtain ⟨B, hNIxyB⟩ := (MI / (J \ {x, y}) \ ({x, y} : set α
 )).exists_base,
  have hNCxyB := hNIxyB,
  rw [contract_delete_comm _ (@disjoint_sdiff_left _ {x, y} J), ←
 union_singleton,
    ← delete_delete, hMy, delete_delete, union_singleton,
```

```
    ← contract_delete_comm _ (@disjoint_sdiff_left _ {x, y} J)] at
 hNCxyB,
have hB : (MI / (J \ {x, y})).base B ↔ (MC / (J \ {x, y})).base B,
{ refine ⟨λ hI, _, λ hC, _⟩,
  -- duplicate code, turn into lemma
  { by_contra h2,
    have hCB := hNCxyB.indep.of_delete,
    obtain ⟨B', hB'⟩ := (MC / (J \ ({x, y} : set α))).exists_base,
    rw [← hI.card] at hNIC,
    apply h2,
    apply hCB.base_of_rk_le_card,
    rw hNIC },
  { by_contra h2,
    have hIB := hNIxyB.indep.of_delete,
    obtain ⟨B', hB'⟩ := (MI / (J \ ({x, y} : set α))).exists_base,
    rw [← hC.card] at hNIC,
    apply h2,
    apply hIB.base_of_rk_le_card,
    rw hNIC } },
by_cases (MI / (J \ {x, y})).base B,
{ apply hNIneNC,
  have hfund : ∀ e ∈ (MI / (J \ {x, y})).E, (MI / (J \ {x,
 y})).fund_circuit e B
    = (MC / (J \ {x, y})).fund_circuit e B,
    intros e he,
    by_cases h2 : e = y,
    { rw h2 at *,
      have h3 : disjoint (insert y B) {x},
        apply disjoint_singleton_right.2 (mem_insert_iff.1.mt _),
        push_neg,
        refine ⟨hxy, _⟩,
        have h10 := hNIxyB.subset_ground,
        rw [delete_ground, ← union_singleton, ← diff_diff] at h10,
        apply not_mem_subset h10 (not_mem_diff_of_mem (mem_singleton
 x)),
      have h5 : disjoint (J \ {x, y}) {x},
        rw [← union_singleton, ← diff_diff],
        apply disjoint_sdiff_left,
      rw [← fund_circuit_delete h.indep (h.mem_cl y) h3,
 MI.contract_delete_comm h5, hMx,
          ← MC.contract_delete_comm h5],
      rw [contract_ground, hIC, ← contract_ground] at he,
```

```
      rw fund_circuit_delete (hB.1 h).indep ((hB.1 h).mem_cl y) h3 },
    { have h3 : disjoint (insert e B) {y},
        apply disjoint_singleton_right.2 (mem_insert_iff.1.mt _),
        push_neg,
        refine ⟨ne.symm h2, _⟩,
        have h10 := hNIxyB.subset_ground,
        rw [delete_ground, ← union_singleton, union_comm, ←
  diff_diff] at h10,
        apply not_mem_subset h10 (not_mem_diff_of_mem (mem_singleton
  y)),
      have h5 : disjoint (J \ {x, y}) {y},
        rw [← union_singleton, union_comm, ← diff_diff],
        apply disjoint_sdiff_left,
      rw [← fund_circuit_delete h.indep (h.mem_cl e) h3,
  MI.contract_delete_comm h5, hMy,
          ← MC.contract_delete_comm h5],
      rw [contract_ground, hIC, ← contract_ground] at he,
      rw fund_circuit_delete (hB.1 h).indep ((hB.1 h).mem_cl e) h3 },
    apply eq_of_forall_fund_circuit_eq φI φC _ h (hB.1 h) hfund,
    simp_rw [contract_ground, hIC] },
  { apply h,
    rw delete_base_iff at hNIxyB hNCxyB,
    cases hiIC with hIc hCc,
    { have h3 := (coindep_contract_iff.2 ⟨hIc, @disjoint_sdiff_right _
  {x, y} J⟩).cl_compl,
      rw ← hNIxyB.cl at h3,
      apply hNIxyB.indep.base_of_cl_eq_ground h3 },
    { have h3 := (coindep_contract_iff.2 ⟨hCc, @disjoint_sdiff_right _
  {x, y} J⟩).cl_compl,
      rw ← hNCxyB.cl at h3,
      apply hB.2,
      apply hNCxyB.indep.base_of_cl_eq_ground h3 } },
end

lemma delete_elem_eq_of_binary {B : set α} {x y : α} (hBxy : (M \ ({x, y}
   : set α)).base B)
  (hBx : (M \ x).base B) (hB : M.base B) [fintype α]
  [module (zmod 2) W] (φ : rep (zmod 2) W (M \ ({x, y} : set α))) {Wx :
   Type*} [add_comm_group Wx]
  [module (zmod 2) Wx]
  (φx : rep (zmod 2) Wx (M \ x)) : (M \ x) =
  (matroid_of_module_fun (zmod 2) (B →₀ zmod 2)
```

```
    (λ e : α, Σ i in (M.fund_circuit e B ∩ B).to_finset, (standard_rep φ
   hBxy) i) M.E) \ x :=
begin
  apply eq_of_indep_iff_indep_forall,
    simp_rw [delete_elem, delete_ground],
    rw matroid_of_module_fun.ground,
    intros I hI,
    rw [(matroid_of_module_fun (zmod 2) (B →₀ zmod 2)
      (λ e : α, Σ i in (M.fund_circuit e B ∩ B).to_finset, (standard_rep
   φ hBxy) i) M.E).delete_elem x,
      delete_indep_iff, ← (standard_rep φx hBx).valid' I hI],
    rw ← (rep_of_matroid_of_module_fun (zmod 2) (B →₀ zmod 2)
      (λ e : α, Σ i in (M.fund_circuit e B ∩ B).to_finset, (standard_rep
   φ hBxy) i) M.E).valid' I _,
    simp [rep_of_matroid_of_module_fun],
    have h12 : (λ (x_1 : α), ite (x_1 ∈ M.E) (Σ (x_1 : α) in
   (M.fund_circuit x_1 B).to_finset
      ∩ B.to_finset, (φ.standard_rep hBxy) x_1) 0) ∘ (coe : I → α) =
      (λ (x_1 : I), ite (x_1.1 ∈ M.E) (Σ (x_1 : α) in (M.fund_circuit
   x_1 B).to_finset
      ∩ B.to_finset, (φ.standard_rep hBxy) x_1) 0),
      simp only [eq_self_iff_true, subtype.val_eq_coe],
    have h10 : ∀ (x_1 : I), ite (x_1.1 ∈ M.E) (Σ (x_1 : α) in
   (M.fund_circuit x_1 B).to_finset
      ∩ B.to_finset, (φ.standard_rep hBxy) x_1) 0 = (Σ (x_1 : α) in
      (M.fund_circuit x_1 B).to_finset ∩ B.to_finset, (φ.standard_rep
   hBxy) x_1),
      { simp only [subtype.val_eq_coe],
        intros e,
        simp_rw [ite_eq_iff],
        left,
        rw delete_elem at hI,
        refine ⟨(M.delete_ground_subset_ground {x}) (hI e.2), rfl⟩ },
    simp_rw [h12, h10],
    have h3 : ((φx.standard_rep hBx) ∘ (coe : I → α)) = λ (e : I),
   ((φx.standard_rep hBx) e),
      simp only [eq_self_iff_true],
    rw [h3],
    simp_rw λ (e : I), (standard_rep φx hBx).mem_sum_basis_zmod2
   hBx.indep _
      (@base.mem_cl _ (M \ x) B hBx e (hI e.2)),
    have hBxs := hBx.subset_ground,
```

```
    simp_rw [delete_elem, delete_ground] at *,
    have h5 := diff_subset M.E {x},
    simp_rw λ (e : I), fund_circuit_delete hB.indep (@base.mem_cl _ M B
   hB e ((diff_subset M.E {x})
      (hI e.2))) (disjoint_singleton_right.2 (mem_insert_iff.1.mt (not_or
   (ne.symm
      (mem_diff_singleton.1 (hI e.2)).2) (not_mem_subset hBxs
      (not_mem_diff_of_mem (mem_singleton x))))))),
    have h6 : (λ (e : ↥I), Σ (x : α) in (M.fund_circuit ↑e B ∩
   B).to_finset, (standard_rep φx hBx) x) =
      (λ (e : ↥I), Σ (x : α) in (M.fund_circuit ↑e B ∩ B).to_finset,
   (standard_rep φ hBxy) x),
      simp only,
      have h10 :=  λ (i : ↥I), @finset.sum_congr _ _ (M.fund_circuit i B ∩
    B).to_finset
        (M.fund_circuit i B ∩ B).to_finset (standard_rep φx hBx)
   (standard_rep φ hBxy) _ rfl _,
      simp_rw  [λ (i : I), h10 i],
      intros x hx,
      rw mem_to_finset at hx,
      have h12 := standard_rep_base_eq φx φ hBx hBxy ⟨x,
   (mem_of_mem_inter_right hx)⟩,
      simp at h12,
      rw h12,
    simp_rw [h6, to_finset_inter, iff_self_and],
    apply λ h, not_mem_subset hI (not_mem_diff_singleton x M.E),
    rw [delete_elem, delete_ground] at hI,
    rw matroid_of_module_fun.ground,
    apply subset_trans hI (diff_subset M.E {x}),
end

end binary_lemmas

section rep_constructions

def rep_empty (𝔽 : Type*) [field 𝔽] (M : matroid α)
  (hM : M.E = ∅) : rep 𝔽 𝔽 M :=
{ to_fun := λ e, 0,
  valid' := λ I hI,
    begin
      rw [hM, subset_empty_iff] at hI,
      rw [hI, @linear_independent_image _ _ _ _ _ _ (∅ : set α) _
```

```
    (inj_on_empty _),
          image_empty],
        simp only [empty_indep, linear_independent_empty 𝔽 𝔽, iff_true]
     end,
  support := λ e he, rfl }

def rep_singleton (𝔽 : Type*) [field 𝔽] (M : matroid α) {x : α} (hMx :
    M.E = {x}) :
  rep 𝔽 𝔽 M :=
{ to_fun := λ e, if hMx : M.nonloop x ∧ e = x then (1 : 𝔽) else (0 : 𝔽),
  valid' := λ I hI,
    begin
      rw hMx at *,
      cases ssubset_or_eq_of_subset hI with hIempty hIsing,
      { rw ssubset_singleton_iff.1 hIempty,
        rw [@linear_independent_image _ _ _ _ _ _ (∅ : set α) _
    (inj_on_empty _),
          image_empty],
        simp only [empty_indep, linear_independent_empty 𝔽 𝔽, iff_true] },
      rw hIsing,
      by_cases M.loop x,
      { have hd : (λ (e : α), dite (M.nonloop x ∧ e = x) (λ (hMx :
    M.nonloop x ∧ e = x), (1 : 𝔽))
          (λ (hMx : ¬(M.nonloop x ∧ e = x)), (0 : 𝔽))) ∘ (coe : ({x} : set
    α) → α)
          = λ x : ({x} : set α), (0 : 𝔽),
            ext;
            simp only [dite_eq_iff],
            right,
            simp_rw not_and_distrib,
            refine ⟨(or.intro_left (¬↑x_1 = x)) h.not_nonloop, rfl⟩,
        rw [hd, ← not_iff_not],
        refine ⟨λ h2, h.dep.not_indep, λ h2, _⟩,
        by_contra,
        apply @linear_independent.ne_zero _ 𝔽 _ ((λ (e : α), (0 : 𝔽)) ∘
    (coe : ({x} : set α) → α))
          _ _ _ _ ⟨x, mem_singleton x⟩ h,
        simp only },
      { have hd : (λ (e : α), dite (M.nonloop x ∧ e = x) (λ (hMx :
    M.nonloop x ∧ e = x), (1 : 𝔽))
          (λ (hMx : ¬(M.nonloop x ∧ e = x)), (0 : 𝔽))) ∘ (coe : ({x} : set
    α) → α)
```

```
      = λ x : ({x} : set α), (1 : 𝔽),
        ext;
        simp only [dite_eq_iff],
        left,
        have h2 := mem_singleton_iff.1 x_1.2,
        simp only [subtype.val_eq_coe] at h2,
        refine ⟨⟨(not_loop_iff (by {rw hMx, apply mem_singleton _})).1
  h, h2⟩, rfl⟩,
      rw hd,
      refine ⟨λ h2, indep_singleton.2 ((not_loop_iff (by {rw hMx, apply
  mem_singleton _})).1 h),
        λ h2, _⟩,
      rw [@linear_independent_image _ _ _ _ _ _ ({x} : set α) (λ e : α
  , (1 : 𝔽))
        (inj_on_singleton _ _), image_singleton],
      apply linear_independent_singleton,
      simp only [ne.def, one_ne_zero, not_false_iff] },
   end,
 support := λ e he,
   begin
     simp only [dite_eq_iff],
     right,
     simp_rw not_and_distrib,
     rw [hMx, mem_singleton_iff] at he,
     refine ⟨(or.intro_right (¬ M.nonloop x)) he, rfl⟩,
   end }

def rep_of_loop (M : matroid α) [finite_rk M] {f : α} (hf : M.loop f)
  (φ : rep 𝔽 W (M \ f)) : rep 𝔽 W M :=
{ to_fun := φ,
  valid' := λ I hI,
    begin
      by_cases f ∈ I,
      { rw ← not_iff_not,
        refine ⟨λ h2, _, λ h2, _⟩,
        { apply indep.nonloop_of_mem.mt,
          simp only [not_forall, exists_prop],
          refine ⟨h, not_nonloop_iff.2 hf⟩ },
        have hfφ := φ.support f,
        by_contra h3,
        have h4 : linear_independent 𝔽 (φ ∘ coe) = linear_independent 𝔽
  (λ (i : I), φ i),
```

```
          simp only [eq_iff_iff],
        rw h4 at h3,
        apply @linear_independent.ne_zero _ 𝔽 W ((λ (i : I), φ i.1)) _ _
    _ _ ⟨f, h⟩ h3,
        simp only,
        apply hfφ,
        rw [delete_elem, delete_ground],
        apply not_mem_diff_singleton },
      have hIf := subset_diff_singleton hI h,
      rw φ.valid,
      simp only [delete_elem, delete_indep_iff, disjoint_singleton_right,
    and_iff_left_iff_imp],
      intros hf2,
      apply h,
    end,
  support := λ e he,
    begin
      by_cases e = f,
      rw h,
      apply φ.support,
      simp only [delete_elem, delete_ground, not_mem_diff_singleton,
    not_false_iff],
      apply φ.support,
      simp only [delete_elem, delete_ground, mem_diff, mem_singleton_iff,
    not_and, not_not],
      contrapose,
      intros,
      apply he
    end }

def add_coloop_rep (φ : rep 𝔽 W M) {f : α} (hf : f ∉ M.E) :
  rep 𝔽 (W × 𝔽) (add_coloop M f) :=
{ to_fun := λ (e : α), if e ∈ ({f} : set α) then linear_map.inr 𝔽 W 𝔽
    ((λ e : α, 1) e) else
    linear_map.inl 𝔽 W 𝔽 (φ e),
  valid' := λ I hI,
    begin
      by_cases f ∈ I,
      { rw [← union_diff_cancel (singleton_subset_iff.2 h), union_comm],
        simp only [← ite_apply _ (linear_map.inr 𝔽 W 𝔽 ∘ (λ e : α, 1))
    (linear_map.inl 𝔽 W 𝔽 ∘ φ)],
        refine ⟨λ h2, _, λ h2, _⟩,
```

```
    { have h11 := linear_independent.image h2,
      rw image_union at h11,
      have hM : M.indep (I \ {f} : set α),
        { have h10 := linear_independent.mono (subset_union_left _ _)
h11,
          rw ← linear_independent_image at h10,
          have h12 : ∀ e : ((I \ {f}) : set α), (ite ((e : α) ∈
({f} : set α))
            ((linear_map.inr 𝔽 W 𝔽) ↑1) ((linear_map.inl 𝔽 W 𝔽) (φ
e))
            = ((linear_map.inl 𝔽 W 𝔽) ∘ φ) e),
          { intros e,
            rw ite_eq_iff,
            right,
            refine ⟨not_mem_of_mem_diff e.2, rfl⟩ },
         simp_rw [λ (e : (I \ {f} : set α)), h12 e,
          @_root_.linear_map.linear_independent_iff _ _ _ _ _ _ _ _
_ _ (linear_map.inl 𝔽 W 𝔽)
          (linear_map.ker_eq_bot_of_injective
linear_map.inl_injective)] at h10,
         rw φ.valid at h10,
         apply h10,
        { intros a ha b hb hab,
          have h13 := h2.injective,
          rw [← restrict_eq, ← inj_on_iff_injective] at h13,
          apply h13 (mem_union_left {f} ha) (mem_union_left {f} hb)
hab } },
        obtain ⟨B2, hB2⟩ := hM,
        rw [← add_coloop_del_eq M hf, delete_elem, delete_base_iff,
add_coloop_ground] at hB2,
        refine ⟨B2 ∪ {f}, ⟨_,
          union_subset_union hB2.2 (subset_refl _)⟩⟩,
        simp only [insert_diff_of_mem, mem_singleton] at hB2,
        rw base_iff_basis_ground,
        have h3 := basis.insert_basis_insert hB2.1 (((add_coloop_eq M
(add_coloop M f) hf).1
          rfl).1.insert_indep_of_indep hB2.1.indep),
        simp only [insert_diff_singleton] at h3,
        rw [add_coloop_ground, union_singleton],
        apply h3 },
    { rw [linear_independent_image, image_union],
      have h12 : (λ (e : α), ite (e ∈ ({f} : set α))
```

```
((linear_map.inr 𝔽 W 𝔽) ↑1)
         ((linear_map.inl 𝔽 W 𝔽) (φ e))) ″ (I \ {f}) =
         (linear_map.inl 𝔽 W 𝔽) ″ (φ ″ (I \ {f})),
         { ext;
           simp only [mem_image, mem_diff, mem_singleton_iff,
comp_app],
           refine ⟨λ h, _, λ h, _⟩,
           { obtain ⟨x, ⟨⟨hx1, hx3⟩, hx2⟩⟩ := h,
             refine ⟨φ x, ⟨⟨x, ⟨⟨hx1, hx3⟩, rfl⟩⟩, _⟩⟩,
             rw [← hx2, eq_comm, ite_eq_iff],
             right,
             refine ⟨hx3, rfl⟩ },
           { obtain ⟨x, ⟨⟨x2, ⟨⟨hx3, hx4⟩, rfl⟩⟩, hx2⟩⟩ := h,
             refine ⟨x2, ⟨⟨hx3, hx4⟩, _⟩⟩,
             rw [← hx2, ite_eq_iff],
             right,
             refine ⟨hx4, rfl⟩ } },
       have h13 : (λ (e : α), ite (e ∈ ({f} : set α))
((linear_map.inr 𝔽 W 𝔽) ↑1)
         ((linear_map.inl 𝔽 W 𝔽) (φ e))) ″ {f} = (linear_map.inr 𝔽 W 𝔽
) ″ (↑1 ″ ({f} : set α)),
         { simp_rw [image_singleton, singleton_eq_singleton_iff,
ite_eq_iff],
           left,
           refine ⟨mem_singleton _, rfl⟩ },
       rw [h12, h13],
       apply linear_independent.inl_union_inr,
       { have h6 := (h2.subset (subset_union_left _
_)).indep_delete_of_disjoint
           (disjoint_sdiff_left),
         rw [← delete_elem, add_coloop_del_eq M hf, ← φ.valid] at h6,
         apply h6.image },
       { rw image_singleton,
         apply linear_independent_singleton,
         simp only [algebra_map.coe_one, pi.one_apply, ne.def,
one_ne_zero, not_false_iff] },
       rw inj_on_union (disjoint_sdiff_left),
       refine ⟨_, ⟨inj_on_singleton _ _, _⟩⟩,
       { intros a ha b hb hab,
         simp only [if_neg (not_mem_of_mem_diff ha), if_neg
(not_mem_of_mem_diff hb)] at hab,
         have hab2 := linear_map.inl_injective hab,
```

```
        have h4 := (h2.subset (subset_union_left _
_)).indep_delete_of_disjoint
          (disjoint_sdiff_left),
        rw [← delete_elem, add_coloop_del_eq M hf] at h4,
        apply (inj_on_of_indep φ h4) ha hb (linear_map.inl_injective
hab) },
      intros a ha b hb,
      simp only [if_pos hb, if_neg (not_mem_of_mem_diff ha)],
      simp only [linear_map.coe_inl, linear_map.coe_inr, ne.def,
prod.mk.inj_iff, not_and],
      intros hc,
      by_contra,
      have h6 := (h2.subset (subset_union_left _
_)).indep_delete_of_disjoint
          (disjoint_sdiff_left),
      rw [← delete_elem, add_coloop_del_eq M hf] at h6,
      apply φ.ne_zero_of_nonloop (h6.nonloop_of_mem ha),
      rw hc } },
  { have h8 : ((λ (e : α), ite (e ∈ ({f} : set α)) ((linear_map.inr 𝔽
 W 𝔽) ↑((λ (e : α), 1) e))
      ((linear_map.inl 𝔽 W 𝔽) (φ e))) ∘ coe) =
      (λ (e : I), ite ((e : α) ∈ ({f} : set α)) ((linear_map.inr 𝔽 W
𝔽) ↑((λ (e : α), 1) e))
      ((linear_map.inl 𝔽 W 𝔽) (φ e))),
      simp only [eq_self_iff_true],
    rw h8,
    have h3 : ∀ (e : I), (ite ((e : α) ∈ ({f} : set α))
      ((linear_map.inr 𝔽 W 𝔽) ↑((λ (e : α), 1) e)) ((linear_map.inl 𝔽
 W 𝔽) (φ e))) =
          ((linear_map.inl 𝔽 W 𝔽) ∘ φ) e,
    { intros,
      simp_rw [ite_eq_iff],
      right,
      refine ⟨mem_singleton_iff.1.mt (ne_of_mem_of_not_mem e.2 h),
rfl⟩ },
    simp_rw [λ (e : I), h3 e],
    rw [@_root_.linear_map.linear_independent_iff _ _ _ _ _ _ _ _ _ _ _
      (linear_map.inl 𝔽 W 𝔽)
      (linear_map.ker_eq_bot_of_injective linear_map.inl_injective), φ
.valid],
    refine ⟨λ h2, _, λ h2, _⟩,
    { rw [← add_coloop_del_eq M hf, delete_elem, delete_indep_iff]
```

```
    at h2,
          apply h2.1 },
        { rw [← add_coloop_del_eq M hf, delete_elem, delete_indep_iff],
          refine ⟨h2, disjoint_singleton_right.2 h⟩ } },
    end,
  support := λ e he,
    begin
      by_cases e ∈ {f},
      { by_contra h2,
        apply he,
        rw [add_coloop_ground, mem_singleton_iff.1 h],
        apply mem_insert },
      { have he2 := not_mem_subset (subset_union_left _ _) he,
        rw ite_eq_iff,
        right,
        refine ⟨h, _⟩,
        simp only [linear_map.coe_inl, prod.mk_eq_zero, eq_self_iff_true,
          and_true],
        rw [add_coloop_ground, mem_insert_iff, not_or_distrib] at he,
        apply φ.support e he.2 },
    end }

def rep_of_del (N : matroid α) (φ : rep 𝔽 W N) (D : set α) :
rep 𝔽 W (N \ D) :=
{ to_fun := λ x, if x ∈ D then 0 else φ.to_fun x,
  valid' := λ I hI, by { rw delete_ground at hI,
    have h2 : ∀ x : I, ite ((x : α) ∈ D) 0 (φ.to_fun x) = φ.to_fun x,
      intros x,
      rw ite_eq_iff,
      right,
      refine ⟨((mem_diff x.1).1 (mem_of_subset_of_mem hI x.2)).2, rfl⟩,
    have h8 : ((λ (e : α), ite ((e : α) ∈ D) 0 (φ.to_fun e)) ∘ coe) =
          (λ (e : I), ite ((e : α) ∈ D) 0 (φ.to_fun e)),
      simp only [eq_self_iff_true],
    rw h8,
    simp_rw [λ (e : I), h2 e],
    refine ⟨λ h, delete_indep_iff.2 ⟨((φ.valid' I (subset_trans hI
   (diff_subset N.E D))).1 h),
    (subset_diff.1 hI).2⟩, λ h, (φ.valid' I (subset_trans hI (diff_subset
   N.E D))).2
      (matroid.delete_indep_iff.1 h).1⟩ },
  support := λ e he,
```

```
    begin
      simp only [ite_eq_iff],
      by_cases e ∈ D,
      left,
      refine ⟨h, rfl⟩,
      right,
      have h2 : e ∉ N.E,
        rw delete_ground at he,
        have h3 : N.E ⊆ (N.E \ D) ∪ D,
          simp only [diff_union_self, subset_union_left],
        apply not_mem_subset h3,
        rw mem_union,
        push_neg,
        refine ⟨he, h⟩,
      refine ⟨h, φ.support e h2⟩,
    end  }

def rep_of_contr (N : matroid α) (φ : rep 𝔽 W N) (C : set α) (hC : C ⊆
   N.E):
  rep 𝔽 (W / span 𝔽 (φ.to_fun '' C)) (N / C) :=
{ to_fun := λ x, submodule.quotient.mk (φ.to_fun x),
  valid' := λ I hI,
    begin
      rw contract_ground at hI,
      have h21 : (λ (x : ↥I), φ.to_fun ↑x) '' univ = φ.to_fun '' I,
        { simp only [to_fun_eq_coe, image_univ],
          ext;
          simp only [mem_range, set_coe.exists, subtype.coe_mk,
   exists_prop, mem_image] },
      obtain ⟨J, hJ⟩ := exists_basis N C hC,
      rw [basis.contract_eq_contract_delete hJ, delete_indep_iff,
        indep.contract_indep_iff hJ.indep],
      have h10 := span_basis φ hJ,
      refine ⟨λ h, _, λ h, _⟩,
      simp only at h,
      simp_rw [← mkq_apply _] at h,
      rw ← φ.valid' _ (union_subset (subset_trans hI (diff_subset _ _))
   hJ.subset_ground_left),
      have h30 : disjoint (span 𝔽 (φ.to_fun '' I)) (span 𝔽 (φ.to_fun ''
   J)),
      { simp_rw [← to_fun_eq_coe] at h10,
        rw h10,
```

```
    simp_rw [← to_fun_eq_coe],
    rw ← ker_mkq (span 𝔽 (φ.to_fun '' C)),
    rw [linear_map.linear_independent_iff, ← image_univ, h21,
disjoint.comm] at h,
    exact h.2 },
  have h7 := linear_independent.image
    (linear_independent.of_comp ((span 𝔽 (φ '' C)).mkq) h),
  have h8 := linear_independent.image ((φ.valid' J
hJ.subset_ground_left).2 (hJ.indep)),
  have h6 := linear_independent.union h7 h8 h30,
  rw [linear_independent_image, image_union],
  refine ⟨⟨_root_.disjoint.of_image (linear_independent.union' h7 h8
h30 h6), h6⟩, _⟩,
  apply @_root_.disjoint.of_image _ _ φ,
  rw disjoint_iff_forall_ne,
  intros x hxI y hyC,
  by_contra h2,
  rw ← h2 at *,
  rw [submodule.disjoint_def, to_fun_eq_coe, h10] at h30,
  specialize h30 x (set_like.mem_coe.1 (mem_of_subset_of_mem
subset_span hxI))
    (set_like.mem_coe.1 (mem_of_subset_of_mem
    (subset_trans (image_subset _ (diff_subset _ _)) subset_span)
hyC)),
  have h31 := mem_of_subset_of_mem
    (image_subset _ (diff_subset _ _)) hyC,
  obtain ⟨e, ⟨he, rfl⟩⟩ := (mem_image φ I x).1 hxI,
  rw to_fun_eq_coe at h7,
  apply @linear_independent.ne_zero _ 𝔽 W _ _ _ _ _ (⟨φ e, hxI⟩ : φ ''
I) h7,
  simp_rw h30,
  simp only [subtype.coe_mk],
  rw inj_on_union (_root_.disjoint.of_image (linear_independent.union'
h7 h8 h30 h6)),
  refine ⟨φ.inj_on_of_indep ((φ.valid' I (subset_trans hI
(diff_subset _ _))).1
    (linear_independent.of_comp ((span 𝔽 (φ '' C)).mkq) h)),
    ⟨φ.inj_on_of_indep (hJ.indep), λ x hx y hy,
set.disjoint_iff_forall_ne.1
    (linear_independent.union' h7 h8 h30 h6) (φ x) (mem_image_of_mem φ
 hx)
    (φ y) (mem_image_of_mem φ hy)⟩⟩,
```

```
    simp_rw [← mkq_apply _],
    rw linear_map.linear_independent_iff,
    refine ⟨(φ.valid' I (indep.subset h.1.2 (subset_union_left I
  J)).subset_ground).2
      (indep.subset h.1.2 (subset_union_left I J)), _⟩,
    rw ker_mkq (span 𝔽 (φ.to_fun '' C)),
    have h60 := linear_independent.image ((φ.valid' _
  h.1.2.subset_ground).2 h.1.2),
    rw image_union at h60,
    rw [← image_univ, h21],
    simp_rw [to_fun_eq_coe],
    simp only [← h10],
    apply linear_independent.union',
    { apply linear_independent.image
        ((φ.valid' J (indep.subset h.1.2 (subset_union_right I
  J)).subset_ground).2
          (indep.subset h.1.2 (subset_union_right I J))) },
    { apply linear_independent.image
        ((φ.valid' I (indep.subset h.1.2 (subset_union_left I
  J)).subset_ground).2
        (indep.subset h.1.2 (subset_union_left I J))) },
    { rw disjoint.comm,
      apply disjoint_image_image,
      have h200 := inj_on_of_indep φ h.1.2,
      rw inj_on at h200,
      intros x hx y hy,
      specialize h200 (mem_of_subset_of_mem (subset_union_left I J) hx)
        (mem_of_subset_of_mem (subset_union_right I J) hy),
      apply mt h200,
      apply disjoint_iff_forall_ne.1 h.1.1 x hx y hy },
    rw [to_fun_eq_coe, union_comm _ _] at h60,
    apply h60,
  end,
support := λ e he,
  begin
    rw contract_ground at he,
    by_cases e ∈ C,
    rw quotient.mk_eq_zero,
    apply mem_of_subset_of_mem subset_span (mem_image_of_mem _ h),
    rw [φ.support, quotient.mk_zero],
    rw ← union_diff_cancel hC,
    apply (mem_union _ _ _).1.mt (not_or_distrib.2 ⟨h, he⟩),
```

```
    end }

def is_rep_of_minor_of_is_rep (N : matroid α) (hNM : N ≤m M) (hM :
   M.is_representable 𝔽) :
  N.is_representable 𝔽 :=
begin
  obtain ⟨B, ⟨hB, ⟨φ⟩⟩⟩ := hM,
  obtain ⟨C, ⟨D, ⟨hC, ⟨hD, ⟨hCD, rfl⟩⟩⟩⟩⟩ :=
   minor.exists_contract_indep_delete_coindep hNM,
  apply is_representable_of_rep (rep_of_del (M / C) (rep_of_contr M φ C
   hC.subset_ground) D),
end

lemma minor_closed_rep : minor_closed (matroid.is_representable 𝔽 :
   matroid α → Prop) :=
  λ M N hNM hM, is_rep_of_minor_of_is_rep N hNM hM

def is_rep_of_iso_minor_of_is_rep (N : matroid γ) (hNM : N ≤i M) (hM :
   M.is_representable 𝔽) :
  N.is_representable 𝔽 :=
begin
  obtain ⟨M', ⟨hM'M, ⟨ψ⟩⟩⟩ := hNM,
  obtain ⟨B, ⟨hB, ⟨φ⟩⟩⟩ := is_rep_of_minor_of_is_rep M' hM'M hM,
  apply is_representable_of_rep (rep_of_iso M' N ψ φ),
end

variables [fintype α]

open_locale big_operators

lemma parallel_extend_rep (φ : rep 𝔽 W M) {x y : α} (hMx : M.nonloop x)
   (hy : y ∉ M.E)
[finite_dimensional 𝔽 W] :
  matroid_of_module_fun 𝔽 W (λ (e : α), if e = y then - φ x else φ e)
   (insert y M.E) =
  parallel_extend M x y :=
begin
  rw ← (eq_parallel_extend_iff hMx hy).2,
  rw circuit_iff_dep_forall_diff_singleton_indep,
    refine ⟨⟨_, λ e he, _⟩, _⟩,
    rw dep,
    refine ⟨_, _⟩,
```

```
 { simp only [matroid_of_module_fun, matroid_of_indep_of_bdd'_apply,
not_and_distrib],
   left,
   --simp_rw [← ite_apply],
   rw not_linear_independent_iff,
   refine ⟨finset.univ, ⟨λ e, 1, _⟩⟩,
   simp only [one_smul, finset.mem_univ, ne.def, one_ne_zero,
not_false_iff, set_coe.exists,
     mem_insert_iff, mem_singleton_iff, exists_prop, and_true,
exists_or_eq_right],
   convert_to (∑ (x_1 : α) in {x, y}, ite (x_1 = y) (-φ x) (φ x_1) =
0),
   rw @finset.sum_subtype _ _ _ ({x, y} : set α) _ {x, y},
   refl,
   intros e,
   rw [← finset.mem_coe, finset.coe_insert, finset.coe_singleton],
   refl,
   rw [finset.sum_insert (finset.mem_singleton.1.mt
(ne_of_mem_of_not_mem hMx.mem_ground hy)),
     finset.sum_singleton, if_pos rfl, if_neg (ne_of_mem_of_not_mem
hMx.mem_ground hy)],
   simp only [add_right_neg] },
 rw [insert_eq, union_comm, ← insert_eq],
 apply insert_subset_insert (singleton_subset_iff.2 hMx.mem_ground),
 obtain ⟨rfl | _⟩ := mem_insert_iff.1 he,
 { simp only [insert_diff_of_mem, mem_singleton,
     diff_singleton_eq_self (mem_singleton_iff.1.mt
(ne_of_mem_of_not_mem hMx.mem_ground hy)),
     matroid_of_module_fun, matroid_of_indep_of_bdd'_apply,
not_and_distrib],
   refine ⟨_, singleton_subset_iff.2 (mem_insert y _)⟩,
   have h2 : ∀ e : ({y} : set α), ↑e = y,
     intros e,
     apply mem_singleton_iff.1 e.2,
   simp_rw [h2, eq_self_iff_true, if_true],
   rw [@linear_independent_image _ _ _ _ _ _ _ (λ (e : α), - φ x)
(inj_on_singleton _ _),
     image_singleton],
   apply @linear_independent_singleton 𝔽 _ _ _ _ _ _ _
     (neg_ne_zero.2 (φ.ne_zero_of_nonloop hMx)) },
 rw [mem_singleton_iff.1 h, insert_eq x {y}, union_comm, ← insert_eq],
 simp only [insert_diff_of_mem, mem_singleton,
```

```
      diff_singleton_eq_self (mem_singleton_iff.1.mt (ne.symm
      (ne_of_mem_of_not_mem hMx.mem_ground hy))), matroid_of_module_fun,
      matroid_of_indep_of_bdd'_apply, not_and_distrib],
    refine ⟨_, singleton_subset_iff.2 (mem_of_mem_of_subset
   hMx.mem_ground (subset_insert y _))⟩,
    have h2 : ∀ e : ({x} : set α), ↑e ≠ y,
      intros e,
      have h3 := (ne_of_mem_of_not_mem hMx.mem_ground hy),
      rw ← mem_singleton_iff.1 e.2 at h3,
      apply h3,
    simp_rw [h2, if_false],
    rw [linear_independent_image (inj_on_singleton _ _), image_singleton],
    exact linear_independent_singleton (φ.ne_zero_of_nonloop hMx),
  simp only [delete_elem, ← delete_matroid_of_module_fun,
   insert_diff_of_mem, mem_singleton,
    diff_singleton_eq_self hy],
  have h10 : ∀ e : α, e ∈ M.E → ite (e = y) (-φ x) (φ e) = φ e,
    intros e he,
    rw if_neg (ne_of_mem_of_not_mem he hy),
  simp_rw [matroid_of_module_fun_congr 𝔽 W _ _ _ h10],
  rw ← matroid_of_module_fun_rep_eq,
end

def series_extend_rep (φ : rep 𝔽 W M) {x y : α} (hx : x ∈ M.E)
  (hy : y ∉ M.E) (hMx : ¬ M.coloop x) : rep 𝔽 (W × 𝔽) (series_extend M x
   y) :=
{ to_fun := λ (e : α),
    if e = x
    then
      (linear_map.inl 𝔽 W 𝔽 ∘ φ + linear_map.inr 𝔽 W 𝔽 ∘ (λ e : α, 1)) e
    else
      if e = y then linear_map.inr 𝔽 W 𝔽 1 else (linear_map.inl 𝔽 W 𝔽 ∘ φ
   ) e,
  valid' := λ I hI,
    begin
      refine ⟨_, λ h2, _⟩,
      { contrapose,
      intros h2,
      rw linear_dependent_comp_subtype',
      rw not_indep_iff at h2,
      obtain ⟨C, ⟨hCI, hCcct⟩⟩ := exists_circuit_subset_of_dep h2,
      by_cases hxC : x ∈ C,
```

```
   { have hyC := (series_pair_mem_circuit _ _ _ hCcct
       (series_extend_cocircuit hx hMx hy)).1 hxC,
     rw [← @union_diff_cancel _ {y} C (singleton_subset_iff.2 hyC),
union_comm,
       union_singleton] at hCcct,
     have hMcct := contract_circuit_of_insert_circuit y (C \ {y})
       ((series_extend_cocircuit hx hMx hy).nonloop_of_mem
       (mem_insert_of_mem x (mem_singleton _)))
(not_mem_diff_singleton _ _) hCcct,
     rw [series_extend_contract_eq M x y hy] at hMcct,
     obtain ⟨f, ⟨hC, ⟨hftot, hfne0⟩⟩⟩ := rep.circuit φ hMcct,
     rw ← hC at hCcct hMcct,
     refine ⟨(⟨(insert y f.support), (λ e : α, if e = y then - f x
else f e), λ a,
       ⟨λ ha, _, λ ha, _⟩⟩ : α →₀ 𝔽), _⟩,
     { obtain ⟨rfl | ha⟩ := finset.mem_insert.1 ha,
       { simp only [eq_self_iff_true, if_true, ne.def, neg_eq_zero],
         rw [← ne.def, ← finsupp.mem_support_iff, ← finset.mem_coe,
hC],
         apply mem_diff_of_mem hxC (mem_singleton_iff.1.mt
(ne_of_mem_of_not_mem hx hy)) },
       { rw if_neg (ne_of_mem_of_not_mem (finset.mem_coe.2 h)
           (not_mem_subset (subset_of_eq hC) (not_mem_diff_singleton _
_))),
         apply finsupp.mem_support_iff.1 h } },
     { apply finset.mem_insert.2,
       by_cases hay : a = y,
       { apply or.intro_left _ hay },
       { rw if_neg hay at ha,
         apply or.intro_right _ (finsupp.mem_support_iff.2 ha) } },
     refine ⟨_, ⟨_, _⟩⟩,
     { rw finsupp.mem_supported,
       simp only [finset.coe_insert, hC],
       apply insert_subset.2 ⟨mem_of_subset_of_mem hCI hyC,
subset_trans (diff_subset _ _) hCI⟩},
     { simp_rw finset.insert_eq y f.support,
       dsimp [finsupp.total_apply, finsupp.sum],
       dsimp [finsupp.total_apply, finsupp.sum] at hftot,
       simp_rw [ite_smul, smul_ite],
       simp only [prod.ext_iff, prod.smul_mk, zero_add, add_zero,
algebra.id.smul_eq_mul,
         mul_one, smul_zero],
```

```
      rw [finset.sum_union, ← @finset.sdiff_union_of_subset _ _ ({x}
: finset α) f.support _,
        finset.sum_union, finset.sum_singleton],
      simp only [if_pos rfl, if_neg (ne_of_mem_of_not_mem hx hy),
        if_neg (ne.symm (ne_of_mem_of_not_mem hx hy)), ←
prod_mk_sum],
      have hx2 : ∀ (e : α), e ∈ ({x} : finset α) → e ≠ y,
        intros e he,
        rw [finset.mem_singleton.1 he],
        apply ne_of_mem_of_not_mem hx hy,
      have hx3 : ∀ (e : α), e ∈ ({x} : finset α) → e = x,
        intros e he,
        rw [finset.mem_singleton.1 he],

      rw [finset.sum_ite_of_false _ _ hx2, finset.sum_ite_of_true _ _
hx3],
      simp only [neg_smul, eq_self_iff_true, if_true, pi.add_apply,
        prod.mk_add_mk, add_zero, zero_add, prod.smul_mk,
algebra.id.smul_eq_mul, mul_one,
        prod.neg_mk],

      simp only [prod.fst_add, zero_add, prod.fst_zero, prod.snd_add,
prod.snd_zero],
      rw [finset.sum_ite_of_false _ _ (λ e he, _),
finset.sum_ite_of_false _ _ (λ e he, _)],
      simp only [finset.sum_ite_of_false _ _ (λ e he, _), ←
prod_mk_sum],
      rw [finset.sum_ite_of_false _ _ (λ e he, _), ← prod_mk_sum,
finset.sum_const_zero,
        zero_add],
      simp only,
      rw ← finset.sum_union, --(finset.sdiff_disjoint),
      simp only [finset.sdiff_union_self_eq_union,
finset.sum_singleton, add_left_neg,
        eq_self_iff_true, and_true],
      rw [finset.union_comm, ← finset.insert_eq,
finset.insert_eq_of_mem],
      apply hftot,
      rw [← finset.mem_coe, hC],
      apply mem_diff_singleton.2 ⟨hxC, ne_of_mem_of_not_mem hx hy⟩,
      simp only [finset.disjoint_singleton_right, finset.mem_sdiff,
finset.mem_singleton,
```

```
        eq_self_iff_true, not_true, and_false, not_false_iff], --
avoiding decidable_eq instance
      rw [← finset.mem_coe, finset.coe_sdiff, hC, mem_diff,
mem_diff] at he,
      apply mem_singleton_iff.2.mt he.1.2,
      rw [finset.mem_sdiff, finset.mem_singleton] at he,
      apply he.2,
      rw [← finset.mem_coe, finset.coe_sdiff, hC, mem_diff,
mem_diff] at he,
      apply mem_singleton_iff.2.mt he.1.2,
      simp only [finset.disjoint_singleton_right, finset.mem_sdiff,
finset.mem_singleton,
        eq_self_iff_true, not_true, and_false, not_false_iff],
      rw [finset.singleton_subset_iff, ← finset.mem_coe, hC],
      apply mem_diff_singleton.2 ⟨hxC, ne_of_mem_of_not_mem hx hy⟩,
      rw [← finset.disjoint_coe, hC],
      simp only [finset.coe_singleton, disjoint_singleton_left,
not_mem_diff_singleton,
        not_false_iff] },
    rw [ne.def, finsupp.ext_iff],
    push_neg,
    use x,
    simp only [ne.def, finsupp.coe_mk, finsupp.coe_zero,
pi.zero_apply],
    rw if_neg (ne_of_mem_of_not_mem hx hy),
    apply finsupp.mem_support_iff.1,
    rw [← finset.mem_coe, hC],
    apply mem_diff_singleton.2 ⟨hxC, ne_of_mem_of_not_mem hx hy⟩ },
  { have hyC := (series_pair_mem_circuit _ _ _ hCcct
      (series_extend_cocircuit hx hMx hy)).2.mt hxC,
    have h4 := (@indep.of_contract _ _ _ {y}).mt (not_indep_iff.2
hCcct.dep),
    rw [← contract_elem, series_extend_contract_eq M x y hy, ← φ
.valid,
      linear_dependent_comp_subtype'] at h4,
    obtain ⟨f, ⟨hC, ⟨hftot, hfne0⟩⟩⟩ := h4,
    refine ⟨f, ⟨subset_trans hC hCI, ⟨_, hfne0⟩⟩⟩,
    dsimp [finsupp.total_apply, finsupp.sum],
    dsimp [finsupp.total_apply, finsupp.sum] at hftot,
    simp_rw smul_ite,
    rw [finset.sum_ite_of_false _ _ (λ e he, _),
      finset.sum_ite_of_false _ _ (λ e he, _)],
```

```
    simp only [prod.smul_mk, algebra.id.smul_eq_mul, mul_zero, ←
prod_mk_sum, hftot,
      finset.sum_const_zero, prod.mk_eq_zero, eq_self_iff_true,
and_self],
    { apply ne_of_mem_of_not_mem (finset.mem_coe.2 he)
        (not_mem_subset ((f.mem_supported _).1 hC) hyC) },
    { apply ne_of_mem_of_not_mem (finset.mem_coe.2 he)
        (not_mem_subset ((f.mem_supported _).1 hC) hxC) } } },
  { simp_rw [linear_independent_comp_subtype, finsupp.total_apply,
smul_ite],
    dsimp [finsupp.sum],
    simp only [add_zero, zero_add, mul_one, smul_zero, mul_zero,
finset.sum_ite, prod.ext_iff,
      finset.filter_congr_decidable, prod.fst_add, prod.fst_zero,
prod.snd_add,
      prod.snd_zero, finset.filter_eq', finset.filter_ne', ←
prod_mk_sum,
      finset.sum_const_zero, zero_add, add_zero],
    intros l hl hl0,
    by_cases hyI : (series_extend M x y).indep ({y} ∪ I : set α),
    { have hyI2 := (hyI.subset (subset_union_left _
_)).union_indep_iff_contract_indep.1 hyI,
      rw [← contract_elem, series_extend_contract_eq M x y hy, ← φ
.valid,
        linear_independent_comp_subtype] at hyI2,
      simp_rw [finsupp.total_apply] at hyI2,
      have hxl : x ∉ l.support,
      { by_contra hxl,
        rw [if_pos hxl] at hl0,
        specialize hyI2 (l.filter (≠ y)) _ _,
        { rw [finsupp.mem_supported, finsupp.support_filter,
finset.filter_ne',
            finset.coe_erase],
          apply diff_subset_diff_left ((l.mem_supported _).1 hl) },
        { rw [finsupp.sum_filter_index, finsupp.support_filter,
finset.filter_ne',
            finset.sum_eq_add_sum_diff_singleton (finset.mem_erase.2
            ⟨ne_of_mem_of_not_mem hx hy, hxl⟩), ← finset.erase_eq],
          rw [finset.erase_right_comm, finset.sum_singleton] at hl0,
          apply hl0.1 },
        apply finsupp.mem_support_iff.1 hxl,
        rw [← l.filter_apply_pos (≠ y) (ne_of_mem_of_not_mem hx hy),
```

```
hyI2],
        simp only [finsupp.coe_zero, pi.zero_apply] },
      simp only [if_neg hxl, finset.sum_empty, zero_add] at hl0,
      have hyl : y ∉ l.support,
      { by_contra hyl,
        rw [if_pos (finset.mem_erase.2 ⟨ne.symm (ne_of_mem_of_not_mem
hx hy), hyl⟩),
          finset.sum_singleton] at hl0,
         apply finsupp.mem_support_iff.1 hyl,
         apply hl0.2 },
      specialize hyI2 l _ _,
      { rw [finsupp.mem_supported],
        apply subset_diff_singleton ((l.mem_supported 𝔽).2 hl) hyl },
      { dsimp [finsupp.sum],
        rw [finset.erase_eq_of_not_mem hxl,
finset.erase_eq_of_not_mem hyl] at hl0,
        apply hl0.1 },
      apply hyI2 },
  { have hyl : y ∉ l.support,
    { by_contra,
      rw [singleton_union, insert_eq_of_mem (mem_of_subset_of_mem
        ((l.mem_supported _).1 hl) h)] at hyI,
      apply hyI h2 },
    rw [if_neg (finset.mem_erase.1.mt (not_and_distrib.2
(or.intro_right _ hyl))),
      finset.sum_empty, add_zero] at hl0,
    have hxl : x ∉ l.support,
    { by_contra hxl,
      simp only [if_pos hxl, finset.sum_singleton] at hl0,
      apply finsupp.mem_support_iff.1 hxl,
      apply hl0.2 },
    rw [if_neg hxl, finset.sum_empty, zero_add] at hl0,
    rw not_indep_iff _ at hyI,
    have hIxy : (series_extend M x y).indep ({y} ∪ (I \ {x}) : set α),
    { by_contra hIyxn,
      obtain ⟨C, ⟨hC, hC2⟩⟩ := exists_circuit_subset_of_dep
((not_indep_iff _).1 hIyxn),
      have hyC : y ∈ C,
      { by_contra hyC,
        rw [singleton_union, subset_insert_iff_of_not_mem hyC] at hC,
        apply hC2.dep.not_indep (h2.subset (subset_trans hC
(diff_subset _ _))) },
```

```
        rw ← series_pair_mem_circuit _ _ _ hC2
    (series_extend_cocircuit hx hMx hy) at hyC,
        apply (not_mem_subset hC ((mem_union _ _ _).1.mt
          (not_or_distrib.2 ⟨mem_singleton_iff.1.mt
    (ne_of_mem_of_not_mem hx hy),
           not_mem_diff_singleton _ _⟩))) hyC,
        apply subset_trans (union_subset_union_right _ (diff_subset I
    {x})) hyI.subset_ground },
      have hyx := (hIxy.subset (subset_union_left _
    _)).union_indep_iff_contract_indep.1 hIxy,
      rw [← contract_elem, series_extend_contract_eq M x y hy, ← φ
    .valid,
        linear_independent_comp_subtype] at hyx,
      rw [finset.erase_eq_of_not_mem hxl, finset.erase_eq_of_not_mem
    hyl] at hl0,
      apply hyx l ((l.mem_supported _).2
        (subset_diff_singleton (subset_diff_singleton ((l.mem_supported
    _).1 hl) hxl) hyl)) hl0.1,
      --apply hyx.subset_ground,
      rw series_extend_ground hx at hI ⊢,
      simp only [singleton_union, auto_param_eq],
      apply insert_subset.2 ⟨mem_insert _ _, hI⟩ } },
    end,
  support := λ e he,
    begin
      rw series_extend_ground hx at he,
      rw [if_neg, if_neg],
      simp only [linear_map.coe_inl, prod.mk_eq_zero, eq_self_iff_true,
    and_true],
      apply φ.support _ (not_mem_subset (subset_insert _ _) he),
      apply ne.symm (ne_of_mem_of_not_mem (mem_insert y _) he),
      apply ne.symm (ne_of_mem_of_not_mem (mem_insert_of_mem _ hx) he),
    end }

end rep_constructions

section unif_rep

lemma U1k_representable (k : ℕ) (hk : 1 ≤ k) [nontrivial 𝔽] : (unif 1
   k).is_representable 𝔽 :=
begin
  have φ := @rep.mk _ 𝔽 _ _ _ _ (unif 1 k) (λ x, (1 : 𝔽)) (λ I hI, _)
```

```
    (by { intros e he,
          by_contra,
          apply he,
          simp only [unif_ground_eq, mem_univ] }),
  { apply is_representable_of_rep φ },
  rw [unif_indep_iff],
  refine ⟨λ h, _, λ h, _⟩,
  { rw [ncard, nat.card_eq_fintype_card, ← finrank_self 𝔽],
    apply fintype_card_le_finrank_of_linear_independent h },
  { cases le_iff_lt_or_eq.1 h with h0 h1,
    { rw [ncard_eq_zero.1 (nat.lt_one_iff.1 h0), linear_independent_image
   (λ x hx y hy hxy,
        (inj_on_empty (λ x, (1 : 𝔽))) hx hy rfl), image_empty],
      apply linear_independent_empty 𝔽 _ },
    { obtain ⟨a, rfl⟩ := ncard_eq_one.1 h1,
      rw [linear_independent_image (λ x hx y hy hxy, (inj_on_singleton (λ
   x, (1 : 𝔽)) a) hx hy rfl),
        image_singleton],
      apply linear_independent_singleton,
      simp only [ne.def, one_ne_zero, not_false_iff] } },
end

lemma U23_binary : matroid.is_binary (unif 2 3) :=
begin
  have hcard3 : fintype.card ((set.univ \ {0}) : set (fin 2 → zmod 2)) =
   3,
  { rw [← to_finset_card, to_finset_diff, finset.card_sdiff,
   to_finset_univ, finset.card_univ,
      to_finset_card, card_singleton, @fintype.card_fun (fin 2) (zmod 2)
   _ _ _, zmod.card 2,
      fintype.card_fin, pow_two, nat.sub_one, nat.pred_eq_succ_iff,
   two_mul],
    simp only [to_finset_univ, to_finset_subset, finset.coe_univ,
   singleton_subset_iff] },
  have f := equiv.symm (fintype.equiv_fin_of_card_eq hcard3),
  have φ := @rep.mk _ (zmod 2) (fin 2 → zmod 2) _ _ _ (unif 2 3) (λ x,
   (f x)) (λ I hI, _)
    (by { simp only [unif_ground_eq, mem_univ, not_true,
   is_empty.forall_iff, forall_const]}),
  { rw [matroid.is_binary, is_representable],
    apply is_representable_of_rep φ },
  rw [unif_indep_iff],
```

```
  refine ⟨λ h, _, λ h, _⟩,
  -- now the possible sizes of vector families for h are 0, 1, 2.
  { rw [ncard, nat.card_eq_fintype_card, ← @finrank_fin_fun (zmod 2) _ _
   2],
    apply fintype_card_le_finrank_of_linear_independent h },
  rw [linear_independent_image (λ x hx y hy hxy,
    (f.injective.inj_on I) hx hy (subtype.coe_inj.1 hxy))],
  cases le_iff_lt_or_eq.1 h with h1 h2,
  cases le_iff_lt_or_eq.1 (nat.le_of_lt_succ h1) with h0 h1,
  { rw [ncard_eq_zero.1 (nat.lt_one_iff.1 h0), image_empty],
    apply linear_independent_empty (zmod 2) _ },

  { obtain ⟨a, rfl⟩ := ncard_eq_one.1 h1,
    rw [image_singleton],
    apply linear_independent_singleton,
    -- if i plug this in directly it wants me to provide a nontrivial
   (zmod 2) instance
    apply (mem_diff_singleton.1 (f a).2).2 },

  { obtain ⟨x, ⟨y, ⟨hxy, rfl⟩⟩⟩ := ncard_eq_two.1 h2,
    rw [image_insert_eq, image_singleton, linear_independent_insert,
   mem_span_singleton, not_exists],
    refine ⟨linear_independent_singleton ((mem_diff_singleton.1 (f
   y).2).2), λ a, _⟩,
    cases le_iff_lt_or_eq.1 (nat.le_of_lt_succ (zmod.val_lt a)) with h0
   h1,
    { rw [(zmod.val_eq_zero a).1 (nat.lt_one_iff.1 h0), zero_smul],
      apply ne.symm (mem_diff_singleton.1 (f x).2).2 },
      rw [← zmod.nat_cast_zmod_val a, h1, algebra_map.coe_one, one_smul],
      by_contra,
      apply hxy (f.injective (subtype.coe_inj.1 (eq.symm h))),
      by_contra,
      apply hxy (mem_singleton_iff.2 (f.injective (subtype.coe_inj.1
   (h)))) },
end

lemma U22_binary : matroid.is_binary (unif 2 2) :=
begin
  have h23 : 2 ≤ 3,
    simp only [nat.bit0_le_bit1_iff],
  apply is_rep_of_iso_minor_of_is_rep (unif 2 2) (unif_iso_minor h23)
   U23_binary,
```

```
end

lemma U24_nonbinary : ¬ matroid.is_binary (unif 2 4) :=
begin
  /- Assume for contradiction that `(unif 2 4)` is representable over `
    zmod 2` -/
  by_contra hrep,
  /- Obtain the representation `φ'` and a `base B` of `(unif 2 4)` -/
  obtain ⟨B, ⟨hB, ⟨φ'⟩⟩⟩ := hrep,
  /- The span of the image of `φ'` on the ground set of `unif 2 4` is a
    submodule of
     `B →₀ zmod 2` -/
  have hsubmodule := @span_mono (zmod 2) _ _ _ _ _ _ (subset_univ (φ' ''
    (unif 2 4).E)),
  rw ← span_span at hsubmodule,
  /- The rank of `B →₀ zmod 2` is 2 -/
  have hrank : (finrank (zmod 2) (B →₀ zmod 2)) = 2,
    rw unif_base_iff at hB,
    simp only [finrank_finsupp, fintype.card_of_finset,
   finset.filter_congr_decidable,
      filter_mem_univ_eq_to_finset, ← hB, ncard_def,
   nat.card_eq_fintype_card],
    simp only [bit0_le_bit0, nat.one_le_bit0_iff, nat.lt_one_iff],
  /- `fin 2` is a basis for `B →₀ zmod 2` -/
  have hbasis := finite_dimensional.fin_basis (zmod 2) (B →₀ zmod 2),
  rw hrank at hbasis,
  haveI : fintype (B →₀ zmod 2),
    apply finsupp.fintype,
  /- `B →₀ zmod 2` has cardinality 4 -/
  have hcard := @module.card_fintype _ (zmod 2) (B →₀ zmod 2) _ _ _ _
   hbasis _ _,
  simp only [zmod.card, fintype.card_fin] at hcard,
  /- Because `unif 2 4` is simple, `φ'` must map its elements to nonzero
    elements of
     `B →₀ zmod 2`. So the cardinality of the image of `φ'` on the ground
    set of
     `unif 2 4` must be at most the cardinality of `B →₀ zmod 2` minus the
    zero vector,
     which is 3 -/
  have hcardimagele3 := fintype.card_le_of_embedding (embedding_of_subset
    _ _
     (subset_trans (φ'.subset_nonzero_of_simple (unif_simple 2 4 rfl.le))
```

```
    (@diff_subset_diff_left _ _ _ ({0} : set (B →₀ zmod 2)) (span_le.1
   hsubmodule)))),
  simp_rw [← to_finset_card, to_finset_diff] at hcardimagele3,
  rw [finset.card_sdiff, span_univ, top_coe, to_finset_univ,
   finset.card_univ, hcard,
    to_finset_card, to_finset_singleton, finset.card_singleton] at
   hcardimagele3,
  /- The cardinality of the image of `φ'` of the ground set of `unif 2 4`
    is 4 -/
  have hcard4 : fintype.card (φ' '' (unif 2 4).E) = fintype.card (fin 4),
  { rw card_image_of_inj_on (φ'.inj_on_ground_of_simple (unif_simple 2 4
   rfl.le)),
    simp only [unif_ground_eq, ← to_finset_card, to_finset_univ,
   finset.card_univ] },
  rw [hcard4, fintype.card_fin, pow_two, two_mul, nat.succ_add_sub_one]
   at hcardimagele3,
  /- `linarith` produces a contradiction from `4 ≤ 3` -/
  linarith,
  simp only [span_univ, top_coe, to_finset_univ, to_finset_subset,
    finset.coe_univ, singleton_subset_iff],
end

end unif_rep

variables [fintype α]

open_locale big_operators

lemma nontrivial_excluded_minor (M : matroid α) [finite_rk M]
  (hM : excluded_minor matroid.is_binary M) : nontrivial M.E :=
begin
  by_contra,
  simp only [nontrivial_coe_sort, not_nontrivial_iff] at h,
  cases h.eq_empty_or_singleton with hempty hsing,
  { apply hM.1 (is_representable_of_rep (rep_empty (zmod 2) M hempty)) },
  { obtain ⟨x, hx⟩ := hsing,
    apply hM.1 (is_representable_of_rep (rep_singleton (zmod 2) M hx)) },
end

-- can remove hxy
lemma excluded_minor_noncoloop (M : matroid α) [finite_rk M]
  (hM : excluded_minor {N : matroid α | N.is_representable 𝔽} M) {y : α}
```

```
   (hf : y ∈ M.E) :
  ¬ M.cocircuit {y} :=
begin
  by_contra hcy,
  have h2 := (dual_circuit_iff_cocircuit.2 hcy).nonempty,
  rw [← ground_inter_left (hcy.subset_ground)] at h2,
  obtain ⟨B, ⟨hB, ⟨φ⟩⟩⟩ := hM.delete_mem h2,
  have hyMy : y ∉ (M / y).E,
      rw [contract_elem, contract_ground],
      apply not_mem_diff_of_mem (mem_singleton _),
    have φM := add_coloop_rep φ hyMy,
    simp only [excluded_minor, mem_minimals_prop_iff] at hM,
  apply hM.1,
  rw [contract_elem, contract_ground, ← delete_ground ] at hyMy,
  rw (add_coloop_eq (M \ {y}) M hyMy).2 ⟨coloop_iff_cocircuit.2 hcy,
    delete_elem M y⟩,
  apply is_representable_of_rep φM,
end
-- can remove hxy
lemma coindep_excluded_minor (M : matroid α)
(hM : excluded_minor {N : matroid α | N.is_representable 𝔽} M) (x y : α)
   (hxy : x ≠ y)
(hx : {x, y} ⊆ M.E)
  : M.coindep {x, y} :=
begin
  by_contra,
  rw coindep_iff_forall_subset_not_cocircuit at h,
  push_neg at h,
  obtain ⟨K, hK1, hK2⟩ := h,
  have h2 := (dual_circuit_iff_cocircuit.2 hK2).nonempty,
  cases ssubset_or_eq_of_subset hK1 with hKs hKeq,
  obtain ⟨a, ⟨ha1, ha2⟩⟩ := ssubset_iff_subset_diff_singleton.1 hKs,
  obtain ⟨rfl | h2⟩ := (mem_insert_iff.1 ha1),
  -- duplicate code
  -- use add_coloop_rep,
  { simp only [insert_diff_of_mem, mem_singleton, diff_singleton_eq_self
      (mem_singleton_iff.1.mt hxy), subset_singleton_iff_eq] at ha2,
    cases ha2 with hempty hs,
    { apply (nonempty_iff_ne_empty.1 h2) hempty },
    rw hs at *,
    apply excluded_minor_noncoloop M hM (singleton_subset_iff.1
   hK2.subset_ground) hK2 },
```

```
{ rw mem_singleton_iff.1 h at *,
  rw [← union_singleton, union_comm, union_singleton] at *,
  simp only [insert_diff_of_mem, mem_singleton, diff_singleton_eq_self
    (mem_singleton_iff.1.mt (ne.symm hxy)), subset_singleton_iff_eq] at
 ha2,
  cases ha2 with hempty hs,
  { apply (nonempty_iff_ne_empty.1 h2) hempty },
  rw hs at *,
  apply excluded_minor_noncoloop M hM (singleton_subset_iff.1
 hK2.subset_ground) hK2 },
rw hKeq at *,
have hyy := singleton_nonempty y,
rw ← ground_inter_left (insert_subset.1 hx).2 at hyy,
--rw [ground_inter_left _] at hyy,
have h3 := hM.contract_mem hyy,
obtain ⟨B, ⟨hB, ⟨φ⟩⟩⟩ := h3,
rw ← M.contract_elem y at φ,
have hxMy : x ∈ (M / y).E,
  rw [contract_elem, contract_ground],
  apply (mem_diff x).2,
  refine ⟨_, mem_singleton_iff.1.mt hxy⟩,
  apply mem_of_subset_of_mem hx,
  simp only [mem_insert_iff, eq_self_iff_true, true_or],
have hyMy : y ∉ (M / y).E,
  rw [contract_elem, contract_ground],
  apply not_mem_diff_of_mem (mem_singleton _),
--have hf := series_extend_eq (M / y) M hK2 hxMy rfl hyMy, math is dumb
  stupid
simp only [excluded_minor, mem_minimals_prop_iff] at hM,
apply hM.1,
have hMx : ¬(M / y).coloop x,
  rw [contract_elem, coloop, contract_dual_eq_dual_delete,
 not_loop_iff, delete_nonloop_iff],
  rw [cocircuit, circuit_iff_dep_forall_diff_singleton_indep] at hK2,
  cases hK2 with hxy2 hin,
  specialize hin y (mem_insert_of_mem _ (mem_singleton y)),
  rw [insert_eq, union_comm, ← insert_eq, insert_diff_of_mem _
 (mem_singleton _),
    diff_singleton_eq_self (mem_singleton_iff.1.mt (ne.symm hxy))] at
 hin,
  refine ⟨indep_singleton.1 hin, mem_singleton_iff.1.mt hxy⟩,
rw [(eq_series_extend_iff hxMy hMx hyMy).2 ⟨hK2, rfl⟩, mem_set_of],
```

```
  obtain φM := series_extend_rep φ hxMy hyMy hMx,
  exact is_representable_of_rep φM,
end

lemma excluded_minor_nonloop (M : matroid α) [finite_rk M]
  (hM : excluded_minor matroid.is_binary M) {f : α} (hf : f ∈ M.E) :
   M.nonloop f :=
begin
  by_contra,
  have hfM : ({f} ∩ M.E).nonempty,
    simp only [ground_inter_left, singleton_nonempty],
  obtain ⟨B, ⟨hB, ⟨φ⟩⟩⟩ := hM.delete_mem hfM,
  simp only [not_nonloop_iff] at h,
  apply hM.1 (is_representable_of_rep (rep_of_loop M h φ)),
end

lemma excluded_minor_nonpara (M : matroid α) [finite_rk M]
  (hM : excluded_minor matroid.is_binary M) {x y : α} (hxy : x ≠ y) :
  ¬ M.circuit {x, y}  :=
begin
  by_contra,
  obtain ⟨B, ⟨hB, ⟨φ⟩⟩⟩ := hM.delete_mem (singleton_inter_nonempty.2
   (mem_of_subset_of_mem
    h.subset_ground (mem_insert_iff.2 (or.intro_right (y = x)
   (mem_singleton y))))),
  have hx : (M \ y).nonloop x,
    rw [delete_elem, delete_nonloop_iff],
    cases circuit_iff_dep_forall_diff_singleton_indep.1 h with hxy2 hin,
    { specialize hin y (mem_insert_of_mem _ (mem_singleton y)),
      rw [insert_eq, union_comm, ← insert_eq, insert_diff_of_mem _
   (mem_singleton _),
        diff_singleton_eq_self (mem_singleton_iff.1.mt (ne.symm hxy))] at
   hin,
      refine ⟨indep_singleton.1 hin, mem_singleton_iff.1.mt hxy⟩ },
    { have hy : y ∉ (M \ y).E,
        rw [delete_elem, delete_ground],
        apply not_mem_diff_singleton,
      obtain φM := parallel_extend_rep φ hx hy,
      simp_rw ← delete_elem at φM,
      rw ← (eq_parallel_extend_iff hx hy).2 ⟨h, rfl⟩ at φM,
      apply hM.1 (is_representable_of_rep (rep_of_congr
   (rep_of_matroid_of_module_fun (zmod 2)
```

```
        (B →₀ zmod 2) (λ (e : α), ite (e = y) (-φ x) (φ e)) (insert y (M
    \ y).E)) φM)) },
end

lemma excluded_minor_simple (M : matroid α) [finite_rk M]
  (hM : excluded_minor matroid.is_binary M) : simple M :=
begin
  apply λ e he f hf, (indep_iff_forall_subset_not_circuit (insert_subset.2
    ⟨he, singleton_subset_iff.2 hf⟩)).2 (λ C hC, _),
  by_cases hef : e = f,
  { rw hef at *,
    rw insert_eq_of_mem (mem_singleton f) at hC,
    cases ssubset_or_eq_of_subset hC with hempty heq,
    { rw ssubset_singleton_iff.1 hempty,
      apply empty_not_circuit },
    { rw [heq, ← loop_iff_circuit],
      apply (excluded_minor_nonloop M hM hf).1 } },
  { cases ssubset_or_eq_of_subset hC with hC2 heq,
    { obtain ⟨x, ⟨hx1, hx2⟩⟩ := ssubset_iff_subset_diff_singleton.1 hC2,
      simp only [mem_insert_iff, mem_singleton_iff] at hx1,
      obtain ⟨rfl | rfl⟩ := hx1,
      { simp only [insert_diff_of_mem, mem_singleton, subset_diff] at hx2,
        cases (subset_iff_ssubset_or_eq.1 hx2.1) with hempty heq,
        { rw ssubset_singleton_iff.1 hempty,
          apply empty_not_circuit },
        { rw [heq, ← loop_iff_circuit],
          apply (excluded_minor_nonloop M hM hf).1 } },
      { rw hx1 at *,
        rw [← union_singleton, union_comm, union_singleton] at hx2,
        simp only [insert_diff_of_mem, mem_singleton,
          subset_diff] at hx2,
        cases (subset_iff_ssubset_or_eq.1 hx2.1) with hempty heq,
        { rw ssubset_singleton_iff.1 hempty,
          apply empty_not_circuit },
        { rw [heq, ← loop_iff_circuit],
          apply (excluded_minor_nonloop M hM he).1 } } },
    rw heq,
    apply excluded_minor_nonpara M hM hef },
end

/-- If `M is an excluded minor for binary representation, then `M` has
    rank 2 -/
```

```
lemma excluded_minor_binary_rk2 (M : matroid α) [finite_rk M]
  (hM : excluded_minor (set_of matroid.is_binary) M) : M.rk = 2 :=
begin
  haveI hME := nontrivial_excluded_minor M hM,
  rw [nontrivial_coe_sort, nontrivial_iff_pair_subset] at hME,
  obtain ⟨x, ⟨y, ⟨hxy1, hxy2⟩⟩⟩ := hME,
  have h2 := coindep_excluded_minor M hM x y hxy1 hxy2,

  have hxyr : matroid.is_binary (M \ ({x, y} : set α)),
    apply excluded_minor.delete_mem hM,
    rw ground_inter_left,
    apply insert_nonempty,

  obtain ⟨B, ⟨hBxy, ⟨φ⟩⟩⟩ := hxyr,

  obtain ⟨Bx, ⟨hBx, ⟨φx⟩⟩⟩ := (((excluded_minor_iff _ (@minor_closed_rep _
   (zmod 2) _)).1 hM).2 x
    (hxy2 (mem_union_left {y} (mem_singleton x)))).2,

  obtain ⟨By, ⟨hBy, ⟨φy⟩⟩⟩ := (((excluded_minor_iff _ (@minor_closed_rep _
   (zmod 2) _)).1 hM).2 y
    (hxy2 (mem_union_right {x} (mem_singleton y)))).2,

  have hB := coindep.base_of_basis_del h2 (delete_base_iff.1 hBxy),

  have hBy : (M \ y).base B,
    rw [delete_elem, delete_base_iff],
    apply hB.basis_of_subset _,
    apply subset.trans,
    apply hBxy.subset_ground,
    rw [delete_ground, ← union_singleton, union_comm, ← diff_diff],
    apply diff_subset_diff_left (diff_subset _ _),
    apply diff_subset M.E ({y} : set α),

  have hBx : (M \ x).base B,
    rw [delete_elem, delete_base_iff],
    apply hB.basis_of_subset _,
    apply subset.trans,
    apply hBxy.subset_ground,
    rw [delete_ground, ← union_singleton, ← diff_diff],
    apply diff_subset_diff_left (diff_subset _ _),
    apply diff_subset M.E ({x} : set α),
```

```
have hMM'E : M.E = (matroid_of_module_fun (zmod 2) (B →₀ zmod 2)
    (λ e : α, Σ i in (M.fund_circuit e B ∩ B).to_finset, (standard_rep
 φ hBxy) i) M.E).E,
  rw matroid_of_module_fun.ground,
have hMM'x := delete_elem_eq_of_binary hBxy hBx hB φ φx,
have hByx := hBxy,
have hxyyx : M \ {x, y} = M \ {y, x},
  rw [← union_singleton, union_comm, union_singleton],
rw [← union_singleton, union_comm, union_singleton] at hByx,
have hMM'y := delete_elem_eq_of_binary hByx hBy hB (rep_of_congr φ
 hxyyx) φy,
have hφ : ∀ (a : α), ((rep_of_congr φ hxyyx).standard_rep hByx) a =
 (φ.standard_rep hBxy) a,
{ intros a,
  rw φ.standard_rep_eq_of_congr hxyyx },
simp_rw [λ (a : α), hφ a] at hMM'y,
have hB' : (matroid_of_module_fun (zmod 2) (B →₀ zmod 2)
    (λ e : α, Σ i in (M.fund_circuit e B ∩ B).to_finset, (standard_rep
 φ hBxy) i) M.E).base B,
{ rw hMM'x at hBx,
  rw hMM'y at hBy,
  rw [base_iff_basis_ground, ← @diff_empty _ (matroid_of_module_fun
 (zmod 2) (B →₀ zmod 2)
    (λ e : α, Σ i in (M.fund_circuit e B ∩ B).to_finset, (standard_rep
 φ hBxy) i) M.E).E,
    ← singleton_inter_eq_empty.2 (mem_singleton_iff.1.mt hxy1),
 diff_inter],
  rw [delete_elem, delete_base_iff] at hBx hBy,
  apply basis.basis_union hBx hBy },
have hMM'r : M.rk = (matroid_of_module_fun (zmod 2) (B →₀ zmod 2)
    (λ e : α, Σ i in (M.fund_circuit e B ∩ B).to_finset, (standard_rep
 φ hBxy) i) M.E).rk,
{ rw [← hB'.card, hB.card] },
have hnxy : ({x, y} : set α).ncard = 2,
  { rw ncard_eq_to_finset_card,
    simp only [finite.to_finset_insert, finite.to_finset_singleton],
    apply finset.card_insert_of_not_mem (finset.not_mem_singleton.2
 hxy1) },
have hMM' : M ≠ (matroid_of_module_fun (zmod 2) (B →₀ zmod 2)
  (λ e : α, Σ i in (M.fund_circuit e B ∩ B).to_finset, (standard_rep φ
 hBxy) i) M.E),
```

```
 { by_contra,
   rw [excluded_minor, mem_minimals_prop_iff] at hM,
   apply hM.1,
   rw [h, mem_def],
   apply is_representable_of_rep (rep_of_matroid_of_module_fun (zmod
2) (B →₀ zmod 2)
     (λ e : α, Σ i in (M.fund_circuit e B ∩ B).to_finset,
(standard_rep φ hBxy) i) M.E) },
 rw [ne.def, eq_iff_indep_iff_indep_forall,
matroid_of_module_fun.ground] at hMM',
 simp only [eq_self_iff_true, true_and, not_forall, exists_prop] at
hMM',
 obtain ⟨Z, ⟨hZM, hZ⟩⟩ := hMM',
 rw [iff_def, not_and_distrib] at hZ,
 push_neg at hZ,
 cases hZ with hMZ hM'Z,
 { have hJZ : ∀ (J : set α), M.indep J → Z ⊆ J → J = {x, y},
   { intros J hMJ hZJ,
     -- duplicate code
     have hZx : x ∈ Z,
       { by_contra,
         have hZs : (M \ x).indep Z,
         { rw [delete_elem, delete_indep_iff],
           refine ⟨hMZ.1, disjoint_singleton_right.2 h⟩ },
         rw [hMM'x, delete_elem] at hZs,
         apply hMZ.2 hZs.of_delete },
     have hZy : y ∈ Z,
       { by_contra,
         have hZs : (M \ y).indep Z,
         { rw [delete_elem, delete_indep_iff],
           refine ⟨hMZ.1, disjoint_singleton_right.2 h⟩ },
         rw [hMM'y, delete_elem] at hZs,
         apply hMZ.2 hZs.of_delete },
     have hZxy := union_subset (singleton_subset_iff.2 hZy)
(singleton_subset_iff.2 hZx),
     rw union_singleton at hZxy,
     by_contra,
     have hJxyM : ((J \ {x, y}) ∩ M.E).nonempty,
     { simp only [ground_inter_left],
       apply nonempty_iff_ne_empty.2,
       apply diff_eq_empty.1.mt,
       by_contra h2,
```

```
        apply h (eq_of_subset_of_subset h2 (subset_trans hZxy hZJ)) },
      obtain ⟨BN, ⟨hBN, ⟨φN⟩⟩⟩ := hM.contract_mem hJxyM,
      have φN' := rep_of_contr _ (rep_of_matroid_of_module_fun (zmod 2)
 (B →₀ zmod 2)
        (λ e : α, Σ i in (M.fund_circuit e B ∩ B).to_finset,
 (standard_rep φ hBxy) i) M.E) (J \ {x, y})
        (by { rw matroid_of_module_fun.ground, apply subset_trans
 (diff_subset _ _)
        hMJ.subset_ground }),
      apply h (indep_eq_doubleton_of_subset M (matroid_of_module_fun
 (zmod 2) (B →₀ zmod 2)
        (λ e : α, Σ i in (M.fund_circuit e B ∩ B).to_finset,
 (standard_rep φ hBxy) i) M.E) hMM'r hMM'E
        x y hxy1 (by { left, apply h2 }) hMM'x hMM'y hZx hZy hMZ.1 hMZ.2
 hZJ hMJ φN φN') },
    obtain ⟨BZ, hBZ⟩ := hMZ.1,
    specialize hJZ BZ hBZ.1.indep hBZ.2,
    rw hJZ at *,
    rw [← hBZ.1.card, hnxy] },
{ have hJZ : ∀ (J : set α), (matroid_of_module_fun (zmod 2) (B →₀ zmod
 2)
    (λ e : α, Σ i in (M.fund_circuit e B ∩ B).to_finset, (standard_rep
 φ hBxy) i) M.E).indep J
    → Z ⊆ J → J = {x, y},
  { intros J hMJ hZJ,
    have hZx : x ∈ Z,
      { by_contra,
      have hZs : ((matroid_of_module_fun (zmod 2) (B →₀ zmod 2)
        (λ e : α, Σ i in (M.fund_circuit e B ∩ B).to_finset,
        (standard_rep φ hBxy) i) M.E) \ x).indep Z,
      { rw [delete_elem, delete_indep_iff],
        refine ⟨hM'Z.1, disjoint_singleton_right.2 h⟩ },
      rw [← hMM'x, delete_elem] at hZs,
      apply hM'Z.2 hZs.of_delete },
    have hZy : y ∈ Z,
      { by_contra,
        have hZs : ((matroid_of_module_fun (zmod 2) (B →₀ zmod 2)
          (λ e : α, Σ i in (M.fund_circuit e B ∩ B).to_finset,
          (standard_rep φ hBxy) i) M.E) \ y).indep Z,
        { rw [delete_elem, delete_indep_iff],
          refine ⟨hM'Z.1, disjoint_singleton_right.2 h⟩ },
        rw [← hMM'y, delete_elem] at hZs,
```

```
        apply hM'Z.2 hZs.of_delete },
    have hZxy := union_subset (singleton_subset_iff.2 hZy)
  (singleton_subset_iff.2 hZx),
    rw union_singleton at hZxy,
    by_contra,
    have hJxyM' : ((J \ {x, y}) ∩ (matroid_of_module_fun (zmod 2) (B →₀
  zmod 2)
          (λ e : α, Σ i in (M.fund_circuit e B ∩ B).to_finset,
          (standard_rep φ hBxy) i) M.E).E).nonempty,
    { simp only [ground_inter_left],
      apply nonempty_iff_ne_empty.2,
      apply diff_eq_empty.1.mt,
      by_contra h2,
      apply h (eq_of_subset_of_subset h2 (subset_trans hZxy hZJ)) },
    obtain ⟨BN, ⟨hBN, ⟨φN⟩⟩⟩ := hM.contract_mem hJxyM',
    have φN' := rep_of_contr _ (rep_of_matroid_of_module_fun (zmod 2)
  (B →₀ zmod 2)
      (λ e : α, Σ i in (M.fund_circuit e B ∩ B).to_finset,
  (standard_rep φ hBxy) i) M.E) (J \ {x, y})
      (by { rw matroid_of_module_fun.ground, apply subset_trans
  (diff_subset _ _)
      hMJ.subset_ground }),
    apply h (indep_eq_doubleton_of_subset (matroid_of_module_fun (zmod
  2) (B →₀ zmod 2)
      (λ e : α, Σ i in (M.fund_circuit e B ∩ B).to_finset,
  (standard_rep φ hBxy) i) M.E) M
      (eq.symm hMM'r) (eq.symm hMM'E) x y hxy1 (by { right, apply h2 })
  (eq.symm hMM'x)
      (eq.symm hMM'y) hZx hZy hM'Z.1 hM'Z.2 hZJ hMJ φN' φN) },
   obtain ⟨BZ, hBZ⟩ := hM'Z.1,
   specialize hJZ BZ hBZ.1.indep hBZ.2,
   rw hJZ at *,
   rw [hMM'r, ← hBZ.1.card, hnxy] },
end

lemma excluded_minor_binary_ncard (M : matroid α) [finite_rk M]
  (hM : excluded_minor (set_of matroid.is_binary) M) : 2 ≤ M.E.ncard :=
by { rw [← excluded_minor_binary_rk2 M hM, rk_def], apply r_le_card }

lemma excluded_minor_binary_unif (hM : excluded_minor matroid.is_binary M)
  (ψ : M ≃i unif 2 M.E.ncard) (h2 : 2 ≤ M.E.ncard) : 4 ≤ M.E.ncard :=
begin
```

```
  cases le_iff_eq_or_lt.1 (excluded_minor_binary_ncard M hM) with h2 h3,
  { by_contra,
    rw ← h2 at ψ,
    obtain ⟨B, ⟨hB, ⟨φ⟩⟩⟩ := U22_binary,
    apply hM.1 (is_representable_of_rep (rep_of_iso _ _ ψ φ)) },
  { cases le_iff_eq_or_lt.1 (nat.add_one_le_iff.2 h3) with h2 h3,
    { by_contra,
      rw ← h2 at ψ,
      obtain ⟨B, ⟨hB, ⟨φ⟩⟩⟩ := U23_binary,
      apply hM.1 (is_representable_of_rep (rep_of_iso _ _ ψ φ)) },
    { apply nat.add_one_le_iff.2 h3 } },
end

/-- If `M` is an excluded minor for binary representation, then `unif 2 4
   ≤i M` -/
lemma excluded_minor_binary (M : matroid α) [finite_rk M]
  (hM : excluded_minor (set_of matroid.is_binary) M) : unif 2 4 ≤i M :=
begin
  obtain ⟨ψ⟩ := (iso_line_iff (excluded_minor_binary_ncard M hM)).2 ⟨
   excluded_minor_simple M hM,
    ⟨excluded_minor_binary_rk2 M hM, ⟨to_finite M.E, rfl⟩⟩⟩,
  apply iso_minor.trans (unif_iso_minor (excluded_minor_binary_unif hM ψ
    (excluded_minor_binary_ncard M hM))) (ψ.symm.trans_iso_minor
   (minor.refl.iso_minor)),
end

/-- If `M` is an excluded minor for binary representation, then `M` is
   isomorphic to
  `unif 2 M.E.ncard` -/
lemma excluded_minor_binary_iso_unif (M : matroid α) [finite_rk M]
  (hM : excluded_minor (set_of matroid.is_binary) M) : nonempty (M ≃i
   (unif 2 M.E.ncard)) :=
(iso_line_iff (excluded_minor_binary_ncard M hM)).2 ⟨
   excluded_minor_simple M hM,
⟨excluded_minor_binary_rk2 M hM, ⟨to_finite M.E, rfl⟩⟩⟩

lemma excluded_minor_binary_ncard4 (hM : excluded_minor matroid.is_binary
   M) : 4 = M.E.ncard :=
begin
  obtain ⟨ψ⟩ := excluded_minor_binary_iso_unif M hM,
  cases le_iff_eq_or_lt.1 (excluded_minor_binary_unif hM ψ
   (excluded_minor_binary_ncard M hM))
```

```
    with h3 h4,
  { apply h3 },
  { by_contra,
    obtain ⟨ψ2⟩ := (iso_line_iff (excluded_minor_binary_ncard M hM)).2 ⟨
   excluded_minor_simple M hM,
     ⟨excluded_minor_binary_rk2 M hM, ⟨to_finite M.E, rfl⟩⟩⟩,
    have h4 := (excluded_minor_iff matroid.is_binary (@minor_closed_rep _
   (zmod 2) _)).1 hM,
    obtain ⟨M', ⟨hM'M, ⟨g⟩⟩⟩ := iso_minor.trans (@unif_iso_minor _ _ 2
     (excluded_minor_binary_unif hM ψ2 (excluded_minor_binary_ncard M
   hM))) (ψ2.symm.iso_minor),
    cases le_iff_eq_or_lt.1 (ncard_le_of_subset hM'M.ground_subset) with
   hcontra hlt,
    { apply h,
     rw [ncard_eq_to_finset_card M.E, finite.to_finset_eq_to_finset,
   to_finset_card,
       ((fintype.bijective_iff_injective_and_card ψ2).1 ψ
   2.bijective).2, ← hcontra,
      ncard_eq_to_finset_card M'.E, finite.to_finset_eq_to_finset,
   to_finset_card M'.E,
       ← ((fintype.bijective_iff_injective_and_card g).1
   g.bijective).2, unif_ground_eq,
       ← to_finset_card univ, to_finset_univ, finset.card_univ,
   fintype.card_fin, unif_ground_eq,
       ← to_finset_card univ, to_finset_univ, finset.card_univ,
   fintype.card_fin] },
    { obtain ⟨e, ⟨heM, heM'⟩⟩ := exists_mem_not_mem_of_ncard_lt_ncard hlt,
     apply U24_nonbinary,
     cases hM'M.minor_contract_or_minor_delete ((mem_diff e).2 ⟨heM,
   heM'⟩) with hMe hMe,
     { obtain ⟨B, ⟨hB, ⟨φ⟩⟩⟩ := is_rep_of_minor_of_is_rep _ hMe (h4.2 e
   heM).1,
       apply is_representable_of_rep (rep_of_iso _ _ g φ) },
     { obtain ⟨B, ⟨hB, ⟨φ⟩⟩⟩ := is_rep_of_minor_of_is_rep _ hMe (h4.2 e
   heM).2,
       apply is_representable_of_rep (rep_of_iso _ _ g φ) } } },
end

lemma excluded_minor_binary_iso_unif24 (M : matroid α) [finite_rk M]
  (hM : excluded_minor (set_of matroid.is_binary) M) : nonempty (M ≃i
   (unif 2 4)) :=
by { rw excluded_minor_binary_ncard4 hM, apply
```

```
   excluded_minor_binary_iso_unif M hM }

/-- `unif 2 4` is an excluded minor for binary representation -/
lemma U24_excluded_minor : excluded_minor (set_of matroid.is_binary)
   (unif 2 4) :=
begin
  apply (excluded_minor_iff (set_of matroid.is_binary) (@minor_closed_rep
   _ (zmod 2) _)).2
    ⟨U24_nonbinary, λ e he, ⟨_, _⟩⟩,
  { obtain ⟨B, ⟨hB, ⟨φc⟩⟩⟩ := @U1k_representable (zmod 2) _ 3 _ _,
    obtain ⟨ψc⟩ := (contract_elem_unif 1 3 e),
    apply is_representable_of_rep (rep_of_iso _ _ ψc φc),
    simp only [one_le_bit1, zero_le'] },
  { obtain ⟨B, ⟨hB, ⟨φc⟩⟩⟩ := @U23_binary,
    obtain ⟨ψc⟩ := (delete_elem_unif 2 3 e),
    apply is_representable_of_rep (rep_of_iso _ _ ψc φc) },
end

/-- If `M` is an excluded minor for binary representation, then `M` is
   isomorphic to `unif 2 4` -/
lemma excluded_minor_binary_iff_iso_unif24 (M : matroid α) [finite_rk M] :
  excluded_minor (set_of matroid.is_binary) M ↔ nonempty (M ≃i (unif 2
   4)) :=
begin
  refine ⟨λ hM, excluded_minor_binary_iso_unif24 M hM, λ hφ, _⟩,
  obtain ⟨φ2⟩ := hφ,
  apply (excluded_minor_iff (set_of matroid.is_binary) (@minor_closed_rep
   _ (zmod 2) _)).2
    ⟨_, λ e he, _⟩,
  { by_contra,
    obtain ⟨B, ⟨hB, ⟨φ24⟩⟩⟩ := h,
    obtain φ := rep_of_iso _ _ φ2.symm φ24,
    apply U24_nonbinary (is_representable_of_rep (rep_of_iso _ _ φ2.symm φ
   24)) },
  have hcoe : (coe : M.E → α)⁻¹' {e} = {(⟨e, he⟩ : M.E)},
  { ext;
    simp only [mem_preimage, mem_singleton_iff],
    refine ⟨λ h, subtype.coe_eq_of_eq_mk h, λ h, by { rw h,
      apply subtype.coe_mk e he }⟩ },
  refine ⟨_, _⟩,
  obtain ⟨B, ⟨hB, ⟨φc⟩⟩⟩ := @U1k_representable (zmod 2) _ 3 _ _,
  obtain ⟨ψc⟩ := (contract_elem_unif 1 3 (φ2 ⟨e, he⟩)),
```

```
  rw [contract_elem, ← image_singleton, ← image_singleton, ← hcoe, ←
    iso.image] at ψc,
  apply is_representable_of_rep (rep_of_iso _ _ (iso.trans (contract_iso φ
    2 {e}) ψc) φc),
  simp only [one_le_bit1, zero_le'],

  obtain ⟨B, ⟨hB, ⟨φd⟩⟩⟩ := U23_binary,
  obtain ⟨ψd⟩ := (delete_elem_unif 2 3 (φ2 ⟨e, he⟩)),
  rw [delete_elem, ← image_singleton, ← image_singleton, ← hcoe, ←
    iso.image] at ψd,
  apply is_representable_of_rep (rep_of_iso _ _ (iso.trans (delete_iso φ2
    {e}) ψd) φd),
end

/-- Tutte's 1958 theorem, which states that a matroid is binary if and
    only if it contains no
  `unif 2 4` minor. -/
theorem binary_iff_no_u24_minor (M : matroid α) [finite_rk M] :
  matroid.is_binary M ↔ ¬ unif 2 4 ≤i M :=
begin
  refine ⟨λ hfM, _, λ h3, (@mem_iff_no_excluded_minor_minor _ M _
    (matroid.is_binary)
    (@minor_closed_rep _ (zmod 2) _)).2 (λ M' hM', _)⟩,
  { by_contra,
    obtain ⟨M', ⟨hM', ⟨ψ⟩⟩⟩ := h,
    apply ((excluded_minor_iff (set_of matroid.is_binary)
    (@minor_closed_rep _ (zmod 2) _)).1
      ((excluded_minor_binary_iff_iso_unif24 M').2 ⟨ψ.symm⟩)).1
      (is_rep_of_minor_of_is_rep _ hM' hfM) },
  { by_contra,
    obtain ⟨ψ⟩ := excluded_minor_binary_iso_unif24 M' hM',
    refine h3 ⟨M', ⟨h, ⟨ψ.symm⟩⟩⟩ },
end

end rep

end matroid
```