LECTURE 3: FORMALISING MATHEMATICS

MODULAR PROOFS IN ISABELLE/HOL

CHELSEA EDMONDS | c.l.edmonds@sheffield.ac.uk

Midlands Graduate School 2025 |

University of Sheffield

1

COURSE OVERVIEW

A practical course on effective use of the Isabelle/HOL proof assistant in mathematics and programming languages

Lectures:

- Introduction to Proof Assistants
- Formalising the basics in Isabelle/HOL
- Introduction to Isar, more types, Locales and Type classes
- Case studies:
  - **Formalising Mathematics: Combinatorics & advanced locale reasoning patterns**
  - Program Verification: Formalising semantics, program properties, and introducing modularity/abstraction.

Example Classes:

- Isabelle exercises based on the previous lecture
- Will be drawing from the existing Isabelle tutorials/Nipkow's Concrete Semantic Book, as well as custom exercises (e.g. for locales).

2

LECTURE 3 OVERVIEW

Modular proofs = an engineering-like approach to formalisation.

Yesterday: Introduction to modular techniques

TODAY:

- Formalisation of mathematics (some more history!)
- Case Study: Formalising combinatorial structures
- Some mathematical background: designs, graphs, hypergraphs.
- Locale reasoning patterns
  - Locale interactions
  - Rewriting
  - Mutual & reverse sublocales
- Proofs with Locales
- Advantages vs Limitations

3

FORMALISATION OF MATHS

4

SOME HISTORY

The Kepler Conjecture (1998)
- Hales et al.
- "The Flyspeck Project"
- Complicated Proof
- Relied on code
- HOL Light/Isabelle
- 2014

Four Colour Theorem (1976)
- Gonthier & Werner
- Relied on code
- Coq
- 2005

Prime Number Theorem (1896)
- Avigad/Harrison
- Significant Theorem
- Isabelle/HOL Light
- 2004

Odd Order Theorem
- Gonthier et al.
- Significant Theorem
- Coq
- 2012

5

MORE RECENT DEVELOPMENTS

Proof assistants are firmly entering the domain of regular mathematicians

- Terrence Tao and Tim Gowers = two field medallists commenting regularly on this.
  - See Tao's discussion: https://terrytao.wordpress.com/wp-content/uploads/2024/03/machine-assisted-proof-notices.pdf
- Lean in particular has managed to create a community of mathematicians using proof assistants that didn't previously exist.
  - In many ways emphasizes the importance of other factors like: community chats, documentation, user interface, online accessibility etc.
- Percentages of proof assistant conference papers on mathematical formalisations is increasing (e.g. ITP/CPP).

6

LIBRARIES ACROSS PROOF ASSISTANTS

- Many proof assistants have substantial libraries in their distribution, as well as separate advanced libraries …
  - **Mizar:** Mizar Mathematical Library - http://mizar.org/library/
  - **Rocq (Coq):** Mathematical Components - https://math-comp.github.io/
  - **Isabelle[HOL]:** Archive of Formal Proofs - https://www.isa-afp.org/topics/
  - **Lean:** mathlib - https://github.com/leanprover-community/mathlib4
- Most older libraries are *not* unified (both an advantage and limitation!)

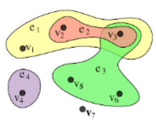
7

A MATHEMATICAL CASE STUDY

COMBINATORIAL STRUCTURES

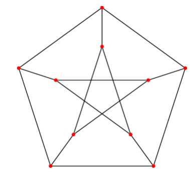
8

MOTIVATING PROBLEM – LARGE HIERARCHIES



- K-uniform hypergraphs
- Regular hypergraphs
- Regular k-uniform graphs
- Non-trivial hypergraphs

- Complete Graphs
- Graph Decompositions (Structure)
- Regular graphs
- Cyclic graphs



Hypergraphs

Graphs

Combinatorial Structures

Designs

Geometric

+ Many more...

Block 1	(C1, D1)	(C2, D2)	(C3, D3)
Block 2	(C1, D2)	(C2, D3)	(C3, D1)
Block 3	(C1, D3)	(C2, D1)	(C3, D2)
Block 4	(C1, D1)	(C2, D3)	(C3, D2)
Block 5	(C1, D2)	(C2, D1)	(C3, D3)
Block 6	(C1, D3)	(C2, D2)	(C3, D1)

- Block designs
- Balanced Designs
- Group Divisible Designs
- Incomplete designs
- Steiner Systems
- Latin Squares

- Projective planes

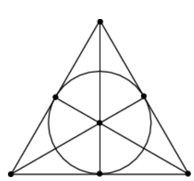
9

THE CHALLENGES

Problem 1:
Many variations and definitions (inconsistent)

Problem 2:
Complex inheritance patterns

Problem 3:
Different language... equivalent structures?



The Fano Plane

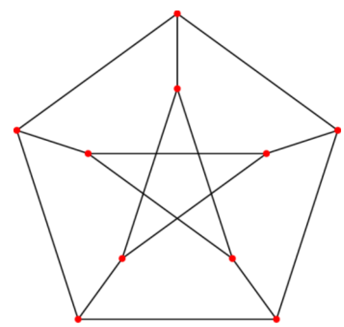
{0, 1, 2}, {0, 3, 4}, {0, 5, 6}, {1, 3, 5}, {1, 4, 6}, {2, 3, 6}, {2, 4, 5}

Design Rep

10

GRAPH THEORY

- There are many known definitions to a (simple) graph:
  - A relation based definition: A set of points V and a well-formed adjacency relation.
  - A set based definition: A set of points V and a set of edges, which are undirected pairs/sets of size two
    - (or just a set of edges, where the vertices are defined implicitly).
- Many types of graphs introducing certain structure
  - Complete Graphs
  - Graph Decompositions (Structure)
  - Regular graphs
  - Cyclic graphs
- Many variations on graphs:
  - Digraphs
  - Multigraphs



11

INTRO TO COMBINATORIAL DESIGNS

"The School Girls Problem (Kirkman, 1850)"

Fifteen young ladies in a school walk out three abreast for seven days in succession: it is required to arrange them daily so that no two shall walk twice abreast.

Sun	Mon	Tue	Wed	Thu	Fri	Sat
ABC	ADG	AEJ	AFO	AHK	AIM	ALN
DEF	BEH	BFL	BDM	BGN	BKO	BIJ
GHI	CJM	CHO	CGL	CFI	CEN	CDK
JKL	FKN	DIN	EIK	DJO	DHL	EGO
MNO	ILO	GKM	HJN	ELM	FGJ	FHM

This is what is known as a 2 – (15, 3, 1) design.

- There are v = 15 *points* – the school girls
- Each block is of size k = 3 – each day the girls are put in groups of 3
- Each pair of points appears together in a block exactly once (λ = 1)

12

INTRO TO COMBINATORIAL DESIGNS

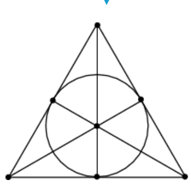
- A design is a finite set of points *V* and a collection of subsets of *V,* called blocks *B (or alternatively, an "incidence relation"*
- Applications range from experimental and algorithm design, to security and communications.
- What makes a design interesting? Properties:
  - The set of block sizes *K*
  - The set of replication numbers *R*
  - The set of t-indices Λ_t
  - The set of intersection numbers *M*
- Language varies: designs, hypergraphs, matrices, geometries, graph decompositions, codes …

13

MORE EXAMPLES

{0, 1, 2}, {0, 3, 4}, {0, 5, 6}, {1, 3, 5}, {1, 4, 6}, {2, 3, 6}, {2, 4, 5}

Design Rep

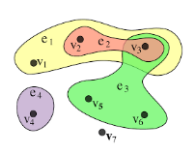


The Fano Plane

- Block size: $k = 3$
- Replication Number: $r = 3$
- Pairwise Points index: $\lambda_2 = 1$
- Intersection Number: $M = \{0, 1\}$

{1, 2, 3}, {2, 3}, {3, 5, 6}, {4}

Design Rep



Hypergraph

- Block size: $K = \{1,2,3\}$
- Replication Number: $R = \{0, 1, 2\}$
- Pairwise Points index: $\Lambda_2 = \{0, 1, 2\}$
- Intersection Number: $M = \{0, 1, 2\}$

14

A BASIC HIERARCHY

COMBINATORIAL DESIGN THEORY

15

INTRODUCING MODULARITY/INHERITANCE: FIRST ATTEMPTS...

Approach 1: Type classes?

```
class incidence_system_class =
  fixes D :: "'a design"
  assumes wellformed: "b ∈# blocks D ⟹ b ⊆ points D"

record 'a block_design = "'a design" +
  size :: "nat"

record 'a balanced_design = "'a design" +
  balance :: "nat"
  t :: nat

record bibd = "'a block_design" + "'a balanced_design"
(* X Can't combine records *)

class block_design = incidence_system_class +
  fixes k :: "nat"
(* X Can't add new type to class *)
```

Approach 2: Records + Locales?

```
record 'a design =
  points :: "'a set "
  blocks :: "'a set multiset"

locale incidence_system =
  fixes D :: "'a design" (structure)
  assumes wf: "b ∈# blocks D ⟹ b ⊆ points D"
```

Messier notation, less automation.

16

THE LOCALE-CENTRIC APPROACH

- Use only locales to model different structures (no complex types/records etc)
- Use *local* definitions inside locale contexts
- Type-synonyms can be used with care to bundle objects
- The "Little Theories" approach for locale definitions (Farmer, 1992).
- Avoid duplication at all costs!
- First Introduced by Ballarin in a paper on "Formalising an Abstract Algebra Textbook" (2020)

```
record 'a design =
  points :: "'a set "
  blocks :: "'a set multiset"

locale incidence_system =
  fixes D :: "'a design" (structure)
  assumes wf: "b ∈# blocks D ⟹ b ⊆ points D"
```

```
locale incidence_system =
  fixes point_set :: "'a set" ("𝒱")
  fixes block_collection :: "'a set multiset" ("ℬ")
  assumes wellformed: "b ∈# ℬ ⟹ b ⊆ 𝒱"
begin
locale design = finite_incidence_system +
  assumes blocks_nempty: "bl ∈# ℬ ⟹ bl ≠ {}"
begin
```

17

THE BASIC DEFINITIONS

```
locale incidence_system =
  fixes point_set :: "'a set" ("𝒱")
  fixes block_collection :: "'a set set" ("ℬ")
  assumes wellformed: "b ∈ ℬ ⟹ b ⊆ 𝒱"
```

```
locale finite_incidence_system = incidence_system +
  assumes finite_sets: "finite 𝒱"
```

```
locale inf_design = incidence_system +
  assumes blocks_nempty: "bl ∈ ℬ ⟹ bl ≠ {}"
```

```
locale design = finite_incidence_system + inf_design
```

Note: These definitions are from a simplified example we'll be exploring in this lecture (no multisets!)

18

AND ANOTHER HIERARCHY....? - HYPERGRAPHS

- Realistically, this is just designs… with another language – so we rename parameters than use direct inheritance!

```
locale hypersystem  =
  fixes vertices :: "'a set" ("𝒱")
  fixes edges  :: "'a set set" ("E")
  assumes wellformed: "b ∈ E ⟹ b ⊆ 𝒱"
```

```
locale fin_hypersystem = hypersystem + finite_incidence_system 𝒱 E
```

```
locale hypergraph = hypersystem + inf_design 𝒱 E
```

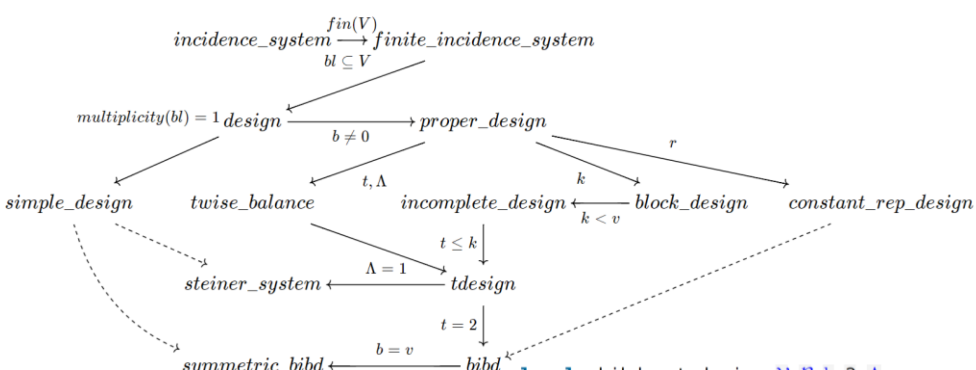
```
locale fin_hypergraph = hypergraph + fin_hypersystem
```

Note: These definitions are from a simplified example we'll be exploring in this lecture (no multisets!)

19

BACK TO THE DESIGN HIERARCHY

- Turns out we can build really big hierarchies!*
- The arrows are annotated with the parameter/assumption added. Dotted arrows indicate indirect inheritance




```
locale t_design = incomplete_design + t_wise_balance +
  assumes block_size_t: "t ≤ k"
```

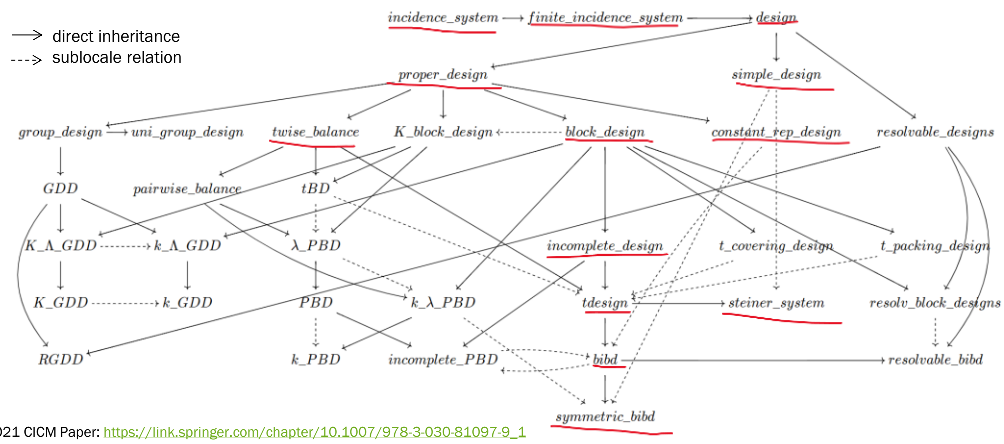
```
locale bibd = t_design 𝒱 ℬ k 2 Λ
  for point_set ("𝒱") and block_collection ("ℬ")
    and u_block_size ("k") and index ("Λ")
```

*Taken from 2021 CICM Paper: https://link.springer.com/chapter/10.1007/978-3-030-81097-9_1

20

EXTENDING THE HIERARCHY

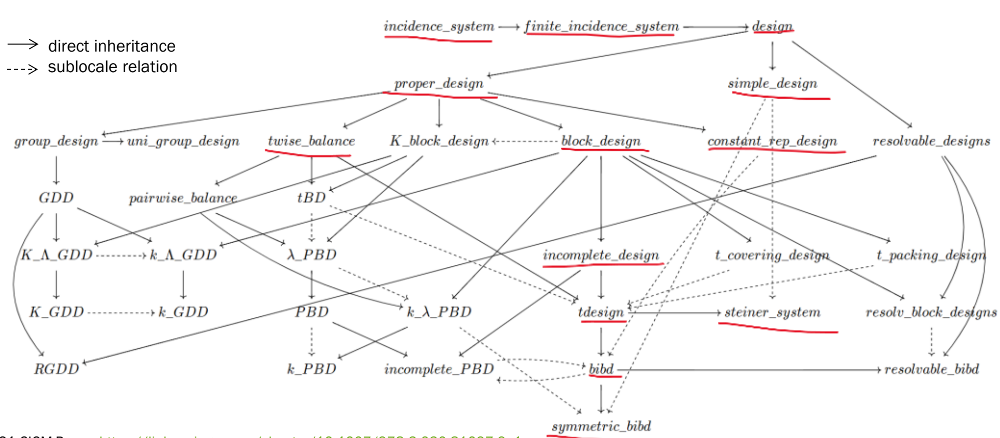
- And expanding it even further!
- Isabelle handles all the relations naturally, but lets zoom in on some of the interesting reasoning patterns



*Taken from 2021 CICM Paper: https://link.springer.com/chapter/10.1007/978-3-030-81097-9_1

EXTENDING THE HIERARCHY

- And expanding it even further!
- Isabelle handles all the relations naturally, but lets zoom in on some of the interesting reasoning patterns



*Taken from 2021 CICM Paper: https://link.springer.com/chapter/10.1007/978-3-030-81097-9_1

LOCALE REASONING PATTERNS

MODELLING INTERACTIONS

23

LOCALE INTERACTIONS – COMBINING LOCALES

- It's always easier to do proofs inside a locale context. So when reasoning on two instances of a locale, why not create another locale? The locale inheritance system allows for such "dual" inheritance

```
locale incidence_system_isomorphism = source: incidence_system 𝒱 ℬ + target: incidence_system 𝒱' ℬ'
  for "𝒱" and "ℬ" and "𝒱'" and "ℬ'" + fixes bij_map ("π")
  assumes bij: "bij_betw π 𝒱 𝒱'"
  assumes block_img: "image_mset ((`) π) ℬ = ℬ'"
begin
lemma design_iso_block_sizes_eq: "source.sys_block_sizes = target.sys_block_sizes"
  apply (simp add: source.sys_block_sizes_def target.sys_block_sizes_def)
  using  design_iso_block_size_eq iso_block_in iso_img_block_orig_exists by force
```

- In a locale with two "instances" of another locale, it is still easy to do proofs using the locale parameters/properties as above (note how **source** and **target** are used)
- But sometimes, we do also want to reason on if two designs are actually isomorphic, without knowing the exact bijection between them.

```
definition isomorphic_systems (infixl "≅D" 50) where
"𝒟 ≅D 𝒟' ⟷ (∃ π . inc_sys_isomorphism (points 𝒟) (blocks 𝒟) (points 𝒟') (blocks 𝒟') π)"
```

24

ASIDE.... A NOTATION TRICK

- When working outside a locale context, sometimes you do want to be able to "bundle parameters". In the isomorphism example below, we're doing this by pair types.

```
definition isomorphic_systems (infixl "≅D" 50) where
"𝒟 ≅D 𝒟' ⟷ (∃ π . inc_sys_isomorphism (points 𝒟) (blocks 𝒟) (points 𝒟') (blocks 𝒟') π)"
```

- Sometimes it's even useful to declare a type synonym to do this.

```
type_synonym 'a design = "'a set × 'a block set"

abbreviation blocks :: "'a design ⇒ 'a block set" where
"blocks D ≡ snd D"
abbreviation points :: "'a design ⇒ 'a set" where
"points D ≡ fst D"
```

More meaningful accessors than **fst** and **snd**

- Generally, you should still avoid doing actual proofs with these types by interpreting the relevant locale as soon as its need.
  - This means you still get all the nice benefits of working with a locale
  - Just with some notational advantages in certain definitions!

25

EQUIVALENT STRUCTURES? - REVERSE SUBLOCALES

- In our hypergraph example, we already connected hypergraphs to designs via direct inheritance.
- But we also want to establish this connection in the opposite direction (i.e. "reverse sublocale")
- And we also want to rewrite block design theorems on certain definitions to use design theoretic language, via the **rewrites** keyword (introduces extra proof goals)

```
sublocale incidence_system ⊆ hypersystem 𝒱 ℬ
  rewrites "hdegree v = rep v" and "hdegree_set vs = index vs"
proof (unfold_locales)
  show "⋀b. b ∈ ℬ ⟹ b ⊆ 𝒱 " using wellformed by simp
  then interpret hs: hypersystem 𝒱 ℬ by (unfold_locales)
  show "hs.hdegree v = rep v"
    using hs.hdegree_def rep_def by simp
  show " hs.hdegree_set vs = index vs"
    using hs.hdegree_set_def index_def by simp
qed
```

Use of rewrites

```
locale hypergraph = hypersystem 𝒱 E + inf_design 𝒱 E
```

reverse sublocale

```
sublocale inf_design ⊆ hypergraph 𝒱 ℬ
  by unfold_locales
```

26

EQUIVALENT STRUCTURES? - MUTUAL SUBLOCALES

- Now consider if we formalised hypergraphs using an incidence relation definition,
- Instead of inheriting directly in one direction, we now need to establish sublocale relationships in *both* directions.

```
locale hypersys_rel =
  fixes vertices :: "'a set" ("V")
  fixes inc_rel :: "('a × 'a set) set" ("I")
  assumes wf: "(v, e) ∈ I   ⟹ v ∈ e ∧ e ⊆ V"
              "(v, e) ∈ I ⟹ (∀ u. u ∈ e ⟶ (u, e) ∈ I)"
```


EQUIVALENT STRUCTURES? - MUTUAL SUBLOCALES

- Let's try it naively....

```
sublocale hypersys_rel ⊆ inf_design V edge_set
  sorry

sublocale inf_design ⊆ hypersys_rel 𝒱 ℐ
  oops
```

Proof state Auto hovering Auto update Update Search:

```
Duplicate constant declaration "local.v" vs. "local.v"
The above error(s) occurred while activating syntax of locale instance
incidence_system "𝒱" "hypersys_rel.edge_set ℐ"
```

- Looping issue! Sublocale loops/naming clashes are the most common issue when establishing this.
- Careful interpretations and rewrites of parameter definitions can help us avoid such loops.

MUTUAL SUBLOCALES

There is a simple 4 step “recipe” for establishing mutual sublocales

1) In each locale, create definitions to represent the any parameters the locales do not share.

inf_design locale

```
definition "𝓘 ≡ { (x, b) . b ∈ 𝓑 ∧ x ∈ b}"
```

hypersys_rel locale

```
definition edge_set:: "'a set set" where
"edge_set ≡ snd ` I"
```

2) Set up a temporary interpretation of the mutual representation in each locale

inf_design locale

```
interpretation hypersys_rel 𝓥 𝓘
  by (fact is_hypersys_rel)
```

hypersys_rel locale

```
interpretation inf_design V edge_set
  by (fact is_inc_sys)
```


MUTUAL SUBLOCALES

There is a simple 4 step “recipe” for establishing mutual sublocales

3) Establish the equivalence of the interpretation’s version of a property, and the local definition

inf_design locale

```
lemma edge_set_is: "𝓑 = edge_set"
```

hypersys_rel locale

```
lemma rel_inc_is: "I = 𝓘"
```

4) Establish the sublocale declaration in both direction with careful use of the **rewrites** command.
   - Note: rewriting is not required if parameters do not need to be “manipulated”

```
sublocale inf_design ⊆ hypersys_rel 𝓥 𝓘
  rewrites "hypersys_rel.edge_set 𝓘 = 𝓑"
  using is_hypersys_rel edge_set_is by simp_all
```

```
sublocale hypersys_rel ⊆ inf_design V edge_set
  rewrites "inf_design.𝓘 edge_set = I"
  using is_inc_sys rel_inc_is by simp_all
```

- If you further refine both locales, further mutual sublocales can usually be established just via step (4)

PROOF PATTERNS

There are two main proof patterns when establishing something is an instance of a locale

(1) Custom introduction rules

- The **intro_locales** tactic isn't particularly usable by itself – unfolding to the *axiomatic* definition of a locale
- If you commonly know you need to establish locale B for something that already satisfies ancestor locale A's assumptions, define a custom introduction rule!
- This can be in a locale context or outside a locale context

```
lemma finite_sysI2[intro]:
  "finite 𝒱 ⟹ incidence_system 𝒱 ℬ ⟹  finite_incidence_system 𝒱 ℬ"
  using incidence_system.finite_sysI by blast
```

Lemma in **finite_incidence_system** context

```
lemma comp_is_fin_sys: "finite_incidence_system 𝒱 (complement_blocks)"
  using complement_blocks_sys finite_sysI2 finite_sets by blast
```

31

PROOF PATTERNS

There are two main proof patterns when establishing something is an instance of a locale

(2) Local interpretation first

- **unfold_locales** unfolds everything! If you're 10 locales deep into a hierarchy this can be a lot, and annoying if you've already shown elsewhere (even in a different theory) that the parameters satisfy a locale part way through that hierarchy
- By applying local interpretation first, Isabelle takes this into account in the local proof context!

```
lemma complement_design:
  assumes "⋀ bl . bl ∈ ℬ ⟹ bl ≠ 𝒱"
  shows "design 𝒱 (ℬᶜ)"
  apply unfold_locales

☑ Proof state ☑ Auto

proof (prove)
goal (3 subgoals):
 1. ⋀b. b ∈ ℬᶜ ⟹ b ⊆ 𝒱
 2. finite 𝒱
 3. ⋀bl. bl ∈ ℬᶜ ⟹ bl ≠ {}
```

```
lemma complement_design:
  assumes "⋀ bl . bl ∈ ℬ ⟹ bl ≠ 𝒱"
  shows "design 𝒱 (ℬᶜ)"
proof -
  interpret fin: finite_incidence_system 𝒱 "ℬᶜ" us
  interpret inf: inf_design 𝒱 "ℬᶜ" using comp_is_i
  show ?thesis apply unfold_locales

☑ Proof state ☑ Auto hovering ☑ Auto upd

proof (prove)
goal:
No subgoals!
```

32

MORE NOTATION TRICKS – REASONING OUTSIDE OF CONTEXT

- In addition to using a locale as a "definition", you can also easily refer to locale definitions and theorems *outside* a locale in your assumptions
- For example, below, we wanted to use the replication number definition to define a definition *outside the locale context*

Pass locale parameter as well

```
definition points_repn :: "'a design ⇒ nat ⇒ 'a set" where
"points_repn D n = {v . incidence_system.rep (blocks D) v = n}"
```

- Note how in addition to the point (which is all we'd need in the locale context), we also need to pass any of the locale parameters used in the definition (in this case, the blocks).
- Where possible – such definitions should be inside the locale context!

33

ISABELLE DEMONSTRATION

(SIMPLIFIED LIBRARY)

34

MORE LOCALES IN PROOFS

TAKEN FROM RESEARCH LIBRARY

35

USING SYMMETRIC INSTANCES

- In mathematics, we often have symmetric properties – where we can choose something "without loss of generality"
- Locales allow us to minimise the amount of repeated work in a proof environment
- This example is in a *bipartite graph* locale context, which has two extra vertex set parameters for the partition
- We show switching this is still a bipartite graph … which makes it easy to avoid repeating long proofs

```
lemma bipartite_sym: "bipartite_graph V E Y X"
  using partition_ne edge_betw all_bi_edges_sym
  by (unfold_locales) (auto simp add: insert_commute)
```

Show switching X and Y is still bipartite

```
lemma edge_size_degree_sumY: "card E = (∑y ∈ Y . degree y)"
proof -
```

Lemma with 7 line proof

```
lemma edge_size_degree_sumX: "card E = (∑y ∈ X . degree y)"
proof -
  interpret sym: fin_bipartite_graph V E Y X
    using fin_bipartite_sym by simp
  show ?thesis using sym.edge_size_degree_sumY by simp
qed
```

Interpret the symmetrical graph

Use the lemma for the "switched" instance

36

MULTIPLE INSTANCES OF STRUCTURE

- Locales still enable natural reasoning when working with lots of instances of a structure!
- In this example, an assumption establishes that each block allows us to construct a valid K-GDD design, then in the proof we interpret it for an arbitrary block!

```
lemma wilsons_construction_proper:
  assumes "card I = w"
  assumes "w > 0"
  assumes "⋀ n. n ∈ 𝒦' ⟹ n ≥ 2"
  assumes "⋀ B . B ∈# ℬ ⟹ K_GDD (B × I) (f B) 𝒦' {{x} × I |x . x ∈ B }"
  shows "proper_design (𝒱 × I) (∑B ∈# ℬ. (f B))" (is "proper_design ?Y ?B")
proof (unfold_locales, simp_all)
  show "⋀b. ∃x∈#ℬ. b ∈# f x ⟹ b ⊆ 𝒱 × I"
  proof -
    fix b
    assume "∃x∈#ℬ. b ∈# f x"
    then obtain B where "B ∈# ℬ" and "b ∈# (f B)" by auto
    then interpret kgdd: K_GDD "(B × I)" "(f B)" 𝒦' "{{x} × I |x . x ∈ B }" using assms by auto
    show "b ⊆ 𝒱 × I" using kgdd.wellformed
      using ‹B ∈# ℬ› ‹b ∈# f B› wellformed by fastforce
  qed
  show "finite (𝒱 × I)" using finite_sets assms bot_nat_0.not_eq_extremum card.infinite by blast
  show "⋀bl. ∃x∈#ℬ. bl ∈# f x ⟹ bl ≠ {}"
```

Interpret instances from assumption.

37

COMBINING LOCALES ACROSS DISCIPLINES

```
locale dependency_graph = sgraph "V :: 'a set set" E + prob_space "M :: 'a measure" for V E M +
  assumes vin_events: "V ⊆ events"
  assumes mis: "⋀ A. A ∈ V ⟹ mutual_indep_set A (V - ({A} ∪ neighborhood A))"
```

- Locales can be combined no matter their "mathematical" context
- This combines probability with graph theory

38

MODULAR PROOF TECHNIQUES

- Combining locales can also prove valuable in the modularisation of proof techniques – the other side to the "software engineering" approach.
- When reasoning on probabilistic structures, I often needed to start a proof by establishing a probability space (lots of formal infrastructure)
- The example shows how all this infrastructure can be combined using a locale
- + allows us to develop proof techniques in a natural locale context, that can then be used repeatedly!

```
locale vertex_colour_space = fin_hypergraph_nt +
  fixes n :: nat (*Number of colours *)
  assumes n_lt_order: "n ≤ order"
  assumes n_not_zero: "n ≠ 0"
begin

definition "MC = uniform_count_measure (𝒞ⁿ)"

lemma space_eq: "space MC = 𝒞ⁿ"
  unfolding MC_def by (simp add: space_uniform_count_measure)

lemma sets_eq: "sets MC =  Pow (𝒞ⁿ)"
  using emeasure_point_measure_finite
  unfolding MC_def by (simp add: sets_uniform_count_measure)

lemma finite_event: "A ⊆ 𝒞ⁿ ⟹ finite A"
  by (simp add: finite_subset vertex_colourings_fin)
                    :
proposition erdos_propertyB:  (*
   assumes "size E < (2^(k - 1))"
   assumes "k > 0" (* Temporary as
   shows "has_property_B"
proof -
 interpret P: vertex_colour_space 𝒱 E 2
   by unfold_locales (auto simp add: order_ge_two)
```

39

LOCALES: ADVANTAGES VS LIMITATIONS

40

OVERVIEW: ADVANTAGES & LIMITATIONS

Advantages

- Facilitates a "little theories" approach
- Removes duplication
- Increases flexibility and extensibility.
- Easy hierarchy manipulation
- Significant notational benefits.
- Proofs became much neater.
- Transfer of properties
- More modular proofs & proof techniques

Limitations

- Lack of automation
- Increasingly complex locale hierarchy, where sublocale relationships must be maintained.
- Using locale specifications outside of a locale context lacks support (Notational etc)
- Can't naturally define definitions involving multiple instances of structures

41

MORE EXAMPLES IN RESEARCH

- More of this work in combinatorial structures
  (beginning here: https://link.springer.com/chapter/10.1007/978-3-030-81097-9_1,
  See AFP entries here: https://www.isa-afp.org/authors/edmonds/)
- The original fundamental work by Ballarin on Algebra
  (https://dl.acm.org/doi/abs/10.1007/s10817-019-09537-9)
- Work on formalising Schemes in Simple Type Theory by Bordg, Paulson, & Li
  (https://arxiv.org/abs/2104.09366)
- Work on formalising omega categories (Bordg & Mateo)
  https://dl.acm.org/doi/abs/10.1145/3573105.3575679

42

NEXT TIME

- Example Class:
  - Extending our graph theory locales from yesterday
  - Connecting graph theory to (simplified) design/hypergraph library
  - Using reasoning patterns such as equivalence
  - Proving more properties/algebraic extensions (optional)
- Next Lecture:
  - Program verification in proof assistants.
  - Introduction to formalising semantics in Isabelle
    - Including more datatypes, inductive definitions, and functions
  - Case studies in locales use with program semantics
    - Introducing abstraction to proofs
    - Modelling program properties.

43