Logical Methods in Computer Science
Volume 20, Issue 4, 2024, pp. 3:1–3:37
https://lmcs.episciences.org/

Submitted Dec. 27, 2023
Published Oct. 04, 2024

FORMALISING THE DOUBLE-PUSHOUT APPROACH TO GRAPH TRANSFORMATION

ROBERT SÖLDNER AND DETLEF PLUMP

University of York, UK
e-mail address: rs2040@york.ac.uk, detlef.plump@york.ac.uk

Abstract. In this paper, we utilize Isabelle/HOL to develop a formal framework for the basic theory of double-pushout graph transformation. Our work includes defining essential concepts like graphs, morphisms, pushouts, and pullbacks, and demonstrating their properties. We establish the uniqueness of derivations, drawing upon Rosen's 1975 research, and verify the Church-Rosser theorem using Ehrig's and Kreowski's 1976 proof, thereby demonstrating the effectiveness of our formalisation approach. The paper details our methodology in employing Isabelle/HOL, including key design decisions that shaped the current iteration. We explore the technical complexities involved in applying higher-order logic, aiming to give readers an insightful perspective into the engaging aspects of working with an Interactive Theorem Prover. This work emphasizes the increasing importance of formal verification tools in clarifying complex mathematical concepts.

Introduction

Formal methods are instrumental in minimizing software defects by rigorously verifying software correctness against their specification. These methods employ mathematical techniques to assure that software implementations align with predefined specifications and requirements, and include model checking, theorem proving, and symbolic interpretation [HR04].

Interactive theorem provers (ITPs), have validated significant mathematical theorems such as the Four Colour Theorem [Gon07], the Prime Number Theorem [ADGR07], and the Kepler Conjecture [HAB$^+$15]. They have also been instrumental in verifying specific algorithms and software components, like in the cases of the seL4 Microkernel [KEH$^+$09] and the CompCert compiler [Ler09]. These instances showcase the efficacy of ITPs in handling complex theories and systems.

Our research strives to rigorously prove fundamental results in the double-pushout approach to graph transformations [EEPT06], contributing to our overarching goal of verifying specific programs in the GP 2 graph programming language [Plu16, CCP22] using the Isabelle proof assistant [NK14].

This paper not only synthesizes previously published material [SP23] but also introduces new essential technical details, including:

Key words and phrases: Double-pushout graph transformation and Isabelle/HOL and Uniqueness of derivations and Church-Rosser theorem.

DOI:10.46298/LMCS-20(4:3)2024

- The formalisation of graphs, morphisms, and rules, including a discussion of our design decisions.
- Additional details on our gluing construction.
- We added additional proofs, such as the fact that pushouts preserve surjectivity.
- We adopted Theorem 4.3 to cover the commutativity of sequentially independent direct derivations.

While earlier efforts like Strecker's [Str18] have applied ITPs to graph transformation, they have not fully explored the extensive theoretical results available for the double-pushout approach. To the best of our knowledge, our work is the first to formalise the foundational results in double-pushout graph transformation. Further details on related work can be found in Section 5.

In the double-pushout approach, our aim is to abstract from specific node and edge identifiers. To achieve this, we introduce independent type variables representing the types of nodes and edges for each graph. This differentiation enables the use of Isabelle's typechecker during development to prevent unintended mix-ups of node or edge identifiers between graphs. However, we encounter a challenge with Isabelle's inability to quantify over new type variables within locale definitions, which complicates the formalisation of the universal properties of pushouts and pullbacks.

To mitigate this issue, we adopt a strategy in our locales where we define node and edge identifiers as natural numbers. This decision, among others such as potentially using a unified type for both node and edge identifiers, necessitates alternate constructions and the consideration of constraints like ensuring a sufficiently large universe of identifiers to execute the disjoint union. We will discuss these critical design choices in the technical Section 2.

To establish the uniqueness of direct derivations, our proof strategy is divided into two distinct stages. Initially, we focus on demonstrating the uniqueness of the pushout complement. Following this, we prove the uniqueness of the pushout object itself, assuming an isomorphic pushout complement. In the first stage, our approach is inspired by Lack and Sobocinski's methodology [LS04]. However, we adapt their method by incorporating the pushout characterisation and the reduced chain condition [EK79], along with utilising the composition and decomposition lemmas for pushouts and pullbacks.

For the proof of the Church-Rosser theorem, we base our methodology on Ehrig and Kreowski's original work [EK76]. Here too, the pushout characterisation and the reduced chain condition is crucial. We are further employing the composition and decomposition lemmas for pushouts and pullbacks. Additionally, we introduce a lemma that facilitates transitioning between pullbacks and pushouts.

While we recognize the potential to generalise our proofs from graph constructions to adhesive categories, this is not our primary objective. Our decision stems from two considerations: firstly, to avoid the complexity involved in handling an abstract class of categories, such as van Kampen squares; and secondly, because our research involves both abstract concepts like pushouts and pullbacks, as well as set-theoretic constructions for these concepts. Providing the corresponding constructions for all adhesive categories would be impractical.

Our long-term goal is to establish a foundation for verifying graph programs in GP 2, a language fundamentally based on the double-pushout approach to graph transformation. We aspire to provide both interactive and automatic tool support for formal reasoning in graph transformation languages. An effective proof assistant for GP 2 will necessitate concrete definitions of graphs, attributes, rules, derivations, and more, which guides our focus towards graph transformation concepts.

The remainder of this paper is organized as follows:

- Section 1 offers a concise introduction to the Isabelle Proof Assistant, focusing on the constructs employed in our research.
- In Section 2, we delve into the fundamentals of DPO graph transformation and our corresponding formalisation in Isabelle. This section covers the formalisation of key concepts such as graphs, morphisms, and rules.
- Section 3 discusses the uniqueness of direct derivations, including a detailed proof of the uniqueness of pushout complements.
- The Church-Rosser theorem, which posits that parallel independent direct derivations can be rearranged to conclude in a common graph, is explored in Section 4.
- Section 5 presents a brief review of related work in the field, providing context and background to our study.
- Finally, Section 6 summarises our key findings and outlines potential avenues for future research.

All formalisation artefacts, including the complete Isabelle theories, are available on GitHub at `https://github.com/UoYCS-plasma/DPO-Formalisation`.

1. Isabelle/HOL

Isabelle is a versatile, interactive theorem prover that operates on the principles of the LCF (Logic for Computable Functions) approach. Central to its design is a compact meta-logical proof kernel responsible for proof checking, a feature that significantly bolsters confidence in the prover's soundness. When referring to Isabelle/HOL, we are discussing the higher-order logic instantiation within Isabelle, widely recognized as the most mature calculus in its suite [PNW19]. This instantiation is characterised by a strong typing system that supports polymorphic, higher-order functions [BH18].

In Isabelle/HOL, type variables are distinctly marked by a preceding apostrophe. For instance, a term `f` of type `'a` is represented as `f :: 'a`. Our formalisation leverages `locales`, a sophisticated mechanism designed for authoring and structuring parametric specifications. A locale encapsulates a set of parameters $(x_1 \dots x_n)$, assumptions $(A_1 \dots A_m)$, and a resulting theorem, expressed as $\bigwedge x_1 \dots x_n . [\![A_1; \dots ; A_m]\!] \Longrightarrow C$. This approach facilitates the effective combination and enhancement of contexts, yielding a representation that is both clear and maintainable. For further details, Ballarin [Bal21] provides an extensive introduction.

Additionally, we employ *intelligible semi-automated reasoning* (Isar), Isabelle's framework for writing structured proofs [Wen99]. Unlike 'apply-scripts', which linearly execute deduction rules, Isar adopts a structured, organized approach. This methodology significantly improves the readability and maintainability of proofs [NK14]. A comprehensive introduction to Isabelle/HOL is available in [NK14].

2. DPO Graph Transformation in Isabelle

Our formalisation of DPO based graph transformations within the Isabelle/HOL proof assistant is centred around *(finite) directed labelled graphs*.

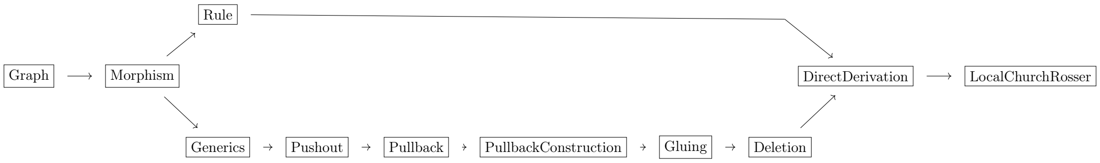

Figure 1. `locale` hierarchy reflecting our formalisation

2.1. **Graphs.** These graphs are defined over an abstract *label alphabet* comprising distinct sets of node and edge labels. We intentionally left labels abstract, as it ensures that the properties we discuss are universally applicable across specific label alphabets. Our definition is inclusive, we accommodate parallel edges and allow loops.

Definition 2.1 (Graph)**.** A *graph* $G = (V, E, s, t, l, m)$ over the alphabet $\mathcal{L}$ is a system where V is the finite set of nodes, E is the finite set of edges, $s, t\colon E \to V$ are functions assigning the source and target to each edge, $l\colon V \to \mathcal{L}_V$ and $m\colon E \to \mathcal{L}_E$ are functions assigning a label to each node and edge. □

The formalisation process within a Proof Assistant mandates adherence to a specific formalism; in our context, this is classical high-order logic. The structure of this work involves a range of design choices. Primarily, we adopt Noschinskis [Nos15] approach, utilizing record types to achieve a concise representation of components, and employing locales to systematically impose properties on these components. Fig. 1 depicts our developed formalisation from the perspective of the locale hierarchy. In this figure, the arrow signifies *used by*. For instance, the `Graph` locale serves as a foundation for the `Morphism` locale. Similarly, `Rule` depends on `Morphism`, and `DirectDerivation` relies on `Rule`.

While portions of our work have been described in preceding work [SP22], this manuscript will refine and update our methodology.

As initially mentioned in the introduction Section, Isabelle/HOL exclusively deals with total functions. This means that these functions are defined across the entire universe of the underlying type. The built-in `option` data type is a standard way to represent optional values, i.e., the presence or absence of data. This sum type can hold two types of values, represented by their constructor:

- `None` used to represent the absence of a value, and
- `Some a` captures the presence of a value, where `a` is the actual value held.

By using the `option` data type, we can simulate partial functions as a function `'a ⇒ 'b option`, which is commonly known as the built-in synonym `map` with the infix ⇀ notation. Another extension found in the standard library are finite sets and finite maps, which are called `fset` and `fmap` respectively. These data types are built on top of the standard (potentially infinite) sets and maps by using the `typedef` keyword, which instructs Isabelle/HOL to provide a new datatype with finitely many elements and the additional finite property.

We consider the following different design options for our formalisation:

(1) The usage of finite maps and sets (`fmap` and `fset`), and
(2) rely on total functions together with the `finite` predicate.

Although from an engineering perspective, the first option to encode properties in the type system seems appealing, it is not without challenges. Here, we can represent graphs using a `pre_graph` record by using the built-in `fmap` and `fset` as follows:

```
record ('v,'e,'l,'m) pre_graph =
  nodes  :: "'v fset"
  edges  :: "'e fset"
  source :: "('e,'v) fmap"
  target :: "('e,'v) fmap"
  node_label :: "('v,'l) fmap"
  edge_label :: "('e,'m) fmap"
```

The implementation of the `locale`, which enforces the corresponding properties of the underlying object, can be expressed using higher abstractions in the context of `fmap` and `fset`. In this scenario, we utilise the built-in native `fmdom` and `fmran` functions to yield a `fset` of the range and domain, mirroring the defined area. This is possible as the finite map relies on the `option` datatype. The integrity of the source function is expressed using `fmdom` $\mathtt{s}_G = \mathtt{E}_G \land$ `fmran` $\mathtt{s}_G$ `|⊆|` $\mathtt{V}_G$. Conversely, for the total function scenario (2), it becomes necessary to explicitly quantify over the set of nodes and edges to convey an equivalent statement. We write $\forall\ \mathtt{v} \in \mathtt{E}_G.\ \ \mathtt{s}_G\ \mathtt{e} \in \mathtt{V}_G$ to express the same property.

Drawing from our experience and evaluations, we chose to further pursue the (2) approach to enhance our automation endeavours. This decision primarily stems from the underdevelopment of `fset`, `map`, and `fmap` theories. We found ourselves in the position of having to establish numerous fundamental properties about `fmap` and `fset` that were not previously proven. Although the Isabelle distributions *lifting and transfer* package [HK13] offered an infrastructure for applying proven properties from underlying theories, such as the `set` and `map` theory, the overall process represented a significant effort. This aspect is further emphasized by Díaz's submissions [Dí20] to the Archive of Formal Proofs (AFP), which enhance the `fmap` theory with new constructs and proven properties.

The corresponding `pre_graph` record is defined by the subsequent Isabelle code.

```
record ('v,'e,'l,'m) pre_graph =
  nodes  :: "'v set"
  edges  :: "'e set"
  source :: "'e ⇒ 'v"
  target :: "'e ⇒ 'v"
  node_label :: "'v ⇒ 'l"
  edge_label :: "'e ⇒ 'm"
```

We present the final locale enforcing the graph properties in the following code block. Notably, there is no need for explicit statements about the properties of node and edge labels, since total functions are defined for the entire universe.

```
locale graph =
  fixes G :: "('v::countable,'e::countable,'l,'m) pre_graph"
  assumes
    finite_nodes: "finite V_G" and
    finite_edges: "finite E_G" and
    source_integrity: "e ∈ E_G ⟹ s_G e ∈ V_G" and
    target_integrity: "e ∈ E_G ⟹ t_G e ∈ V_G"
```

The usage of the built-in `countable` typeclass for node and edge identifiers in our work addresses the Isabelle/HOL constraint against introducing new type variables within a locales definition. This is a result of the simple type theory used by the HOL family [HUW14]. This method parallels the approach found in [SW09], where Isabelle's `statespace` command was used to create an environment with axioms for converting concrete types to natural numbers. Our approach differs by utilizing the injective `to_nat` function provided by the `countable` typeclass, thus enabling efficient type conversion. This specific limitation is further detailed in [SP22]. The injectivity also implies the existence of the inverse, `from_nat`. We use this technique to translate arbitrary identifiers for nodes and edges into a generic representation based on natural numbers.

Definition 2.2 (Graphs over naturals)**.** A graph whose node and edge identifiers are natural numbers is called a *natural graph*.

In Isabelle/HOL, we establish a type synonym `ngraph`, which specializes the existing `pre_graph` structure to `nat`, Isabelle's native type for natural numbers, for both identifiers.

```
type_synonym ('l,'m) ngraph = "(nat,nat,'l,'m) pre_graph"
```

The use of Isabelle/HOL's built-in functions `to_nat` and `from_nat` allows us to convert between both representations. We define the conversion function `to_ngraph` from `ngraph` to the parameterised `pre_graph` structure as follows:

```
definition to_ngraph
  :: "('v::countable,'e :: countable,'l,'m) pre_graph
   ⇒ ('l,'m) ngraph" where
<to_ngraph G ≡ ⦇nodes  = to_nat ` V_G
              ,edges  = to_nat ` E_G
              ,source = λe. to_nat (s_G (from_nat e))
              ,target = λe. to_nat (t_G (from_nat e))
              ,node_label = λv. l_G (from_nat v)
              ,edge_label = λe. m_G (from_nat e)⦈>
```

We convert the set of node and edge identifiers by applying the `to_nat` function, which the `countable` typeclass provides, to each element in these sets using the `image` function, denoted by the `. The source and target functions initially map from `nat` to the original identifier, then apply the source and target functions, and subsequently map back into the `nat` space. For both labelling functions, we simply convert back the `nat` identifiers before applying the corresponding labelling function.

The reverse process, converting a graph over naturals to a parameterized graph, functions similarly. Its important to note that `from_nat (to_nat x) = x` holds true, but the converse does not apply. The crucial fact to remember is that the definition of `to_nat` depends on Hilbert's choice operator `SOME`, which selects a fixed yet arbitrary value. This reliance is a key aspect to consider, as it fundamentally influences the behaviour and limitations of the `to_nat` function. By utilizing `SOME`, `to_nat` essentially encapsulates a degree of arbitrariness, making its outcomes and their relation to the inverse process non-trivial and significant in our context. Consequently, we often need to include type annotations to avoid the inference of different types, which would result in the selection of a different value. This becomes particularly crucial in scenarios involving nested function applications. In these cases, while the final type aligns correctly, the intermediate type might not, leading to potential discrepancies. By actively managing these type annotations, we ensure consistency and accuracy in our function applications, especially when dealing with complex nested structures where type alignment is critical for the intended outcomes.

```
definition from_ngraph
  :: " ('l,'m) ngraph
  ⇒ ('v::countable,'e :: countable,'l,'m) pre_graph" where
‹from_ngraph G ≡ ⦇nodes = from_nat ` V_G
                  ,edges = from_nat ` E_G
                  ,source = λe. from_nat (s_G (to_nat e))
                  ,target = λe. from_nat (t_G (to_nat e))
                  ,node_label = λe. l_G (to_nat e)
                  ,edge_label = λe. m_G (to_nat e)⦈›
```

2.2. Morphisms.

Following this, we will define mappings between graphs that preserve their structure, known as graph morphisms.

Definition 2.3 (Graph morphism)**.** A *graph morphism* $f\colon G \to H$ is a pair of mappings $f = (f_V\colon V_G \to V_H,\ f_E\colon E_G \to E_H)$, such that for all $e \in E_G$ and $v \in V_G$:

(1) $f_V(s_G(e)) = s_H(f_E(e))$ (sources are preserved)
(2) $f_V(t_G(e)) = t_H(f_E(e))$ (targets are preserved)
(3) $l_G(v) = l_H(f_V(v))$ (node labels are preserved)
(4) $m_G(e) = m_H(f_E(e))$ (edge labels are preserved) □

We adopt the same design pattern for graph morphisms as we did for graphs: First, we define a record type to bundle the relevant components, which here include the node and edge mappings. Then, we establish a locale to enforce the properties of the morphism. The morphism record is defined as follows.

```
record ('v_1,'v_2,'e_1,'e_2) pre_morph =
  node_map :: "'v_1 ⇒ 'v_2"
  edge_map :: "'e_1 ⇒ 'e_2"
```

It is important to emphasize that a morphism can map between two different types for each node and edge identifier. This aspect is vital for our subsequent work on the pushout and pullback construction, which we will discuss later in this section.

The morphism locale utilizes the hierarchical inheritance feature of locales, enabling us to depend on the graph locale for defining the source and target graphs. We enhance its functionality by fixating the `pre_morph` record, accomplished using the `fixes` keyword. This is followed by stating the locale axioms, which assert that graphs represent a structure-preserving mapping. This design choice not only streamlines the locale's structure but also ensures a logical integration of graph properties within the morphism context.

```
locale morphism =
  G: graph G +
  H: graph H for
    G :: "('v_1::countable,'e_1::countable,'l,'m) pre_graph" and
    H :: "('v_2::countable,'e_2::countable,'l,'m) pre_graph" +
  fixes
    f :: "('v_1,'v_2,'e_1,'e_2) pre_morph"
  assumes
    morph_edge_range: "e ∈ E_G ⟹ f_E e ∈ E_H" and
    morph_node_range: "v ∈ V_G ⟹ f_V v ∈ V_H" and
    source_preserve : "e ∈ E_G ⟹ f_V (s_G e) = s_H (f_E e)" and
    target_preserve : "e ∈ E_G ⟹ f_V (t_G e) = t_H (f_E e)" and
    label_preserve  : "v ∈ V_G ⟹ l_G v = l_H (f_V v)" and
    mark_preserve   : "e ∈ E_G ⟹ m_G e = m_H (f_E e)"
```

In this context, the graph `G` serves as the source, while `H` represents the target graph. The record encapsulating the node and edge mappings between `G` and `H` is assigned the name `f`. The two assumptions, namely `morph_edge_range` and `morph_node_range`, ensure that for every edge `e` in the source graph, $\mathtt{f}_E$ maps to a corresponding edge in the target graph, and similarly for nodes. The assumptions `source_preserve` and `target_preserve` are critical for maintaining the structural integrity of the graph; they guarantee that the source and target structures are consistently upheld in the mapping process. Lastly, the assumptions `label_preserve` and `mark_preserve` are instrumental in ensuring that the node and edge labels are retained during the mapping, thereby preserving the complete informational content of the graphs.

Definition 2.4 (Special morphisms and isomorphic graphs)**.** A morphism f is *injective* (*surjective*, *bijective*) if f_V and f_E are injective (surjective, bijective). Morphism f is an *inclusion* if for all $v \in V_G$ and $e \in V_E$, $f_V(v) = v$ and $f_E(e) = e$. A bijective morphism is an *isomorphism*. In this case, G and H are *isomorphic*, which is denoted by $G \cong H$. □

Our characterization also takes advantage of the locale inheritance mechanism. An injective morphism inherits all properties from the morphism locale and additionally asserts that both node and edge mappings are injective over the respective sets of nodes and edges of the source graph. We achieve this by employing the built-in `inj_on` predicate. This predicate plays a crucial role in ensuring the uniqueness of each element's mapping in the source graph to the target graph, thereby preserving the distinctiveness of graph elements in the transformation process.

```
locale injective_morphism = morphism +
  assumes
    inj_nodes: "inj_on f_V V_G" and
    inj_edges: "inj_on f_E E_G"
```

It is important to note that the $\mathtt{f}_V$ ($\mathtt{f}_E$) mappings and graph G, with its corresponding components ($\mathtt{V}_G$ and $\mathtt{E}_G$) which are utilized in the body of the `injective_morphism` locale, are also supplied by the `morphism` locale.

A surjective morphism also adheres to this pattern. In this case, we declare that for each node (or edge) in the target graph `H`, there exists a corresponding node (or edge) in the source graph `G` that is mapped to it through the respective morphism function ($\mathtt{f}_V$ and $\mathtt{f}_E$). This condition ensures that every element in the target graph has a pre-image in the source graph, signifying that the mapping covers the entire target graph.

```
locale surjective_morphism = morphism +
  assumes
    surj_nodes: <v ∈ V_H ⟹ ∃v' ∈ V_G. f_V v' = v> and
    surj_edges: <e ∈ E_H ⟹ ∃e' ∈ E_G. f_E e' = e>
```

A bijective morphism combines the principles of both injective and surjective morphisms. In this framework, we assert that for every node (or edge) in the source graph `H`, there is a unique corresponding node (or edge) in the target graph `H`. We choose not to inherit from both the `injective_morphism` and `surjective_morphism` locales. Instead, we utilize the built-in predicate `bij_betw`. A key reason for this decision is to leverage automation: in our experiments, the use of the existing set of proven lemmas significantly facilitated the process of discharging theorems. While it was feasible to demonstrate all these properties if we inherit these locales, we opted to capitalize on the existing infrastructure.

```
locale bijective_morphism = morphism +
  assumes
    bij_nodes: "bij_betw f_V V_G V_H" and
    bij_edges: "bij_betw f_E E_G E_H"
```

Moving forward, we introduce the concept of composing two morphisms to form a singular, joint morphism.

Definition 2.5 (Morphism composition)**.** Let $f\colon F \to G$ and $g\colon G \to H$ be graph morphisms. The *morphism composition* $g \circ f\colon F \to H$ is defined by $g \circ f = (g_V \circ f_V, g_E \circ f_E)$. □

In Isabelle/HOL, we define this composition with a definition that typically hides the underlying details and often requires explicit unfolding or an established set of proven properties. Choosing total functions enables us to use standard function composition techniques for both the node and edge mappings.

```
definition morph_comp
  :: "('v₂,'v₃,'e₂,'e₃) pre_morph
  ⇒  ('v₁,'v₂,'e₁,'e₂) pre_morph
  ⇒('v₁,'v₃,'e₁,'e₃) pre_morph" (infixl "∘→" 55) where
"g ∘→ f = ⦇node_map = g_V ∘ f_V, edge_map = g_E ∘ f_E⦈"
```

Please note that each morphism can alter the typing of the node and edge identifiers. Therefore, in our definition, we introduced three typing parameters. Additionally, we have introduced the infix syntax $\circ_{\rightarrow}$ for graph morphism composition as a form of syntactic sugar.

Lemma 2.6 (Well-definedness of morphism composition)**.** *Given two morphisms $f\colon G \to H$ and $g\colon H \to K$, the composition $g \circ f\colon G \to K$ is also a morphism.*

If we have two valid morphisms, say $f : G \to H$ and $g : H \to K$, then their composition $g \circ f : H \to K$ is also a valid morphism. In Isabelle, we capture this property using the `lemma` keyword along with a specific name (`wf_morph_comp`) for future reference.

```
lemma wf_morph_comp:
  assumes
    f: ‹morphism G H f› and
    g: ‹morphism H K g›
  shows ‹morphism G K (g ∘→ f)›
```

We start by assuming the existence of two valid morphisms, which we name f and g, within the `assumes` blocks. Then, by using the `shows` keyword, we set our target for proof in Isabelle. Following this setup, Isabelle requires the user to interactively work towards achieving the stated goal, essentially proving the validity of the morphism composition.

We open the proof using the `intro_locales` tactic, which will apply the corresponding introduction rule, resulting in the following proof state.

```
proof (state)
goal (3 subgoals):
 1. graph G
 2. graph K
 3. morphism_axioms G K (g ∘→ f)
```

The output shows that we must discharge three subgoals. The first two, which states that G and K are valid graphs, are trivial due to the morphism properties. Since the morphism locale inherits the graph properties for both the source and target graphs, these subgoals follow directly.

Using Isabelle's *Isar* language, we streamline our proof writing process by focusing on the logical structure and progression of arguments. Isar, emphasizing the *what* over the *how*, allows us to structure complex proofs into understandable segments. We initiate each proof by specifying our goal with the `show` keyword, clearly setting our target. This approach not only simplifies the proof of our statement, which in this case follows from the morphism axioms of f and g, but also keeps the proof readable and aligned with traditional mathematical reasoning, enhancing its clarity and verifiability.

To tackle the third subgoal, we focus on the `morphism_axioms` predicate, a product of the locales infrastructure based on the properties we stated. To discharge this subgoal, we initiate the proof using the `standard` tactic, which will execute a single elimination or introduction rule according to the topmost logical connective involved [Wen].

Consequently, our task involves discharging the six morphism axioms stated earlier, which we will briefly outline below. The initial two goals address the fact that for each edge (and node) in `G`, the image created by the composition of the morphisms falls within the subset of the edges (nodes) of `K`.

```
show <morphism_axioms G K (g ∘→ f)>
proof standard
  show <g ∘→ f_E e ∈ E_K> if <e ∈ E_G> for e
    by (simp add: morph_comp_def
                  morphism.morph_edge_range[OF g]
                  morphism.morph_edge_range[OF f]
                  that)
```

Isabelle's simplifier is able to successfully discharges all goals, provided it is equipped with the necessary lemmas. Specifically, for our case, we need to unfold the `morph_comp` definition and, to discharge the first goal, we need to employ the `morph_edge_range` fact for each of the graphs `G` and `K`.

Similarly, the property asserting that the composed morphism preserves the structure under the source function is discharged. This property is defined in the morphism locale (`source_preserve`). We address it using a method similar to the previous approach, employing relevant tactics and lemmas (in a forward directed style using the `OF` keyword).

```
show <g ∘→ f_V (s_G e) = s_K (g ∘→ f_E e)> if <e ∈ E_G> for e
  by (simp add: morph_comp_def
        morphism.morph_edge_range[OF f]
        morphism.morph_edge_range[OF g]
        morphism.source_preserve[OF f]
        morphism.source_preserve[OF g] that)
```

Finally, the preservation of node and edge labels by the morphism follows a similar pattern and is subsequently shown for completeness.

```
show <l_G v = l_K (g ∘→ f_V v)> if <v ∈ V_G> for v
  by (simp add: morph_comp_def
      morphism.label_preserve[OF f]
      morphism.label_preserve[OF g]
      morphism.morph_node_range[OF f] that)
```

The purpose of the earlier discussion was to offer readers an intuitive understanding of how properties are articulated and to provide a glimpse into the overall setup in Isabelle. Moving forward, the focus will shift to describing the fundamental constructs of our formalisation, highlighting selected content that is crucial to understanding the framework and methodology employed.

Please note, if a morphism is uniquely identified by its source and target, we sometimes omit the name and write $F \to G \to H$ to stand for the composition $g \circ f$. In our formalisation, we denote morphism composition using the $\circ_{\rightarrow}$ symbol to prevent a naming clash with Isabelle's built-in function composition.

2.3. **Rules.**

In DPO-based graph transformation, graph morphisms are used to define rules as the atomic units of computation.

Definition 2.7 (Rule)**.** A *rule* $(L \leftarrow K \to R)$ consists of graphs L, K and R over $\mathcal{L}$ together with injective morphisms $K \to L$ and $K \to R$. □

Following the same pattern as before, we use a `pre_rule` record and establish a `rule` locale to enforce the required properties.

```
record ('v1,'e1,'v2,'e2,'v3,'e3,'l,'m) pre_rule =
  lhs    :: "('v1,'e1,'l,'m) pre_graph"
  interf :: "('v2,'e2,'l,'m) pre_graph"
  rhs    :: "('v3,'e3,'l,'m) pre_graph"
```

The polymorphic `pre_rule` record already contains eight type parameters, allowing different node and edge identifiers for each graph of the rule (left-hand side, right-hand side, and the interface) while keeping the labelling types consistent. This setup offers significant flexibility by enabling morphisms between different node and edge representations, which is particularly useful in the gluing and deletion construction. However, this approach comes with a technical drawback: the need to propagate type annotations throughout the codebase. In some cases, this can result in code that is not immediately obvious and may be difficult to read. To address this issue and strike a balance between flexibility and readability, we have reduced the number of locale parameters in our current design by incorporating record types for rules, graphs, morphisms, and other components. This has significantly decreased the number of parameters passed directly into the locales compared to earlier versions of our formalisation.

Despite the challenges, we believe that our current setup offers a sweet spot in terms of conciseness while still maintaining the necessary formal precision. We will explore this trade-off between flexibility and readability in more detail later in the paper, particularly in the context of the gluing and deletion construction.

The locale itself inherits the properties of injective morphisms twice, from the interface to both the left-hand side and the right-hand side.

```
locale rule =
  k: injective_morphism "interf r" "lhs r" b +
  r: injective_morphism "interf r" "rhs r" b'
  for r :: "('v1::countable,'e1::countable
           ,'v2::countable,'e2::countable
           ,'v3::countable,'e3::countable
           ,'l,'m) pre_rule" and b b'
```

We need to populate the `countable` typeclass restriction to allow conversion of graphs to graphs over naturals. The main reason for this is a limitation of the locale mechanism,

$$\begin{tikzcd}
K \arrow[r, "b"] \arrow[d, "d"'] & R \arrow[d, "h"] \\
D \arrow[r] & H
\end{tikzcd}$$

Figure 2. Gluing diagram

preventing the introduction of new type variables in its definition. Once we covered pushouts (and pullbacks), we will briefly continue this discussion.

The addition of graph components along a common subgraph is called *gluing*.

2.4. Gluings.

The gluing construction below employs the disjoint union of sets A and B, typically defined as

$$A + B = (A \times \{1\}) \cup (B \times \{2\}).$$

It incorporates injective functions $i_A \colon A \to A + B$ and $i_B \colon B \to A + B$, ensuring that $i_A(A) \cup i_B(B) = A + B$ and $i_A \cap i_B = \emptyset$.

This does not directly translate into a higher-order logic based proof assistant, as the formalism is not expressive enough to capture the change of representation. Here, we would try to mix $'a$ `set`, where $'a$ is a type variable of the elements, to elements of the type $('a,$ `nat`$)$.

To simplify Lemma 2.8, we assume the sets of D and $R - b(K)$ are disjoint. This assumption prevents the contamination of i_A and i_B usage throughout the lemma while maintaining respect to the more general statement which does not require the disjointness.

Lemma 2.8 (Gluing [Ehr79])**.** *Let $b \colon K \to R$ and $d \colon K \to D$ be injective graph morphisms. Then the following defines a graph H (see Fig. 2), the gluing of D and R according to d:*

(1) $V_H = V_D + (V_R - b_V(V_K))$

(2) $E_H = E_D + (E_R - b_E(E_K))$

(3) $s_H(e) = \begin{cases} s_D(e) & \textit{if } e \in E_D \\ d_V(b_V^{-1}(s_R(e))) & \textit{if } e \in E_R - b_E(E_K) \textit{ and } s_R(e) \in b_V(V_K) \\ s_R(e) & \textit{otherwise} \end{cases}$

(4) t_H *analogous to* s_H

(5) $l_H = \begin{cases} l_D(v) & \textit{if } v \in V_D \\ l_R(v) & \textit{otherwise} \end{cases}$

(6) m_H *analogous to* l_H

*Moreover, the morphism $D \to H$ is an inclusion and the injective morphism h is defined for all items x in R by $h(x) =$ **if** $x \in R - b(K)$ **then** x **else** $d(b^{-1}(x))$.*

Our formalisation of the gluing concept begins with the definition of the corresponding locale, which inherits the two injective morphisms $b \colon K \to R$ and $d \colon K \to D$, as follows.

```
locale gluing =
  d: injective_morphism K D d +
  r: injective_morphism K R b
  for K D R d b
```

Note, the properties of the `injective_locale` for $K \to R$ and $K \to D$ are bound to d and r, respectively. These names are chosen arbitrary and are used to refer to facts of the corresponding morphism, while the arguments to the locale correspond to the morphism objects.

Within the locales context, we present the construction described in Lemma 2.8. Initially, we establish abbreviations, which unlike definitions, do not introduce new concepts that have to be unrolled; instead, they simply allow us to refer to their content by an abbreviated name. This construction relies on the `sum` type, featuring two constructors: `Inl` and `Inr`. We utilize the built-in function `Plus`, denoted by the infix notation `<+>`, for calculating the disjoint sum of sets. This definition relies on applying the `Inl` constructor to each element of the first set and the `Inr` constructor to each element of the second. It is important to note that this setup alters the type; we move from the universe, say `'a` of the first set, and `'b` of the second, to the sum type `'a + 'b`. This shift facilitates the straightforward construction of the disjoint sum, but will result in some complications of our formalisation. We considered imposing the constraint that the nodes (edges) must be disjoint as an alternative approach to the locale. However, to align with our long-term goal of constructing concrete gluings, we chose not to limit our formalisation and proceeded directly with the construction.

Following the gluing construction (cf. Lemma 2.8), we use `V` (and `E`) to represent the elements of the gluing graph.

```
abbreviation V where <V ≡ V_D <+> (V_R - b_V ‘ V_K)>
abbreviation E where <E ≡ E_D <+> (E_R - b_E ‘ E_K)>
```

The ‘ notation refers to the built-in `image` definition, which applies the supplied function to each element of the set. The source function now utilizes the two constructors to employ standard pattern matching for identifying the origin graph of an edge. In the first scenario, with `s (Inl e)`, given the construction of `V` and `E`, it becomes evident that this edge remains within graph `D`. Consequently, we use the source from `D`, repositioning it into the new type via the application of `Inl`. This repositioning reflects the application of the previously mentioned injective function i_A, see above. In the second case, when the source of the edge originates from `R` without the interface, it is necessary to determine whether the source of the edge projects into the interface. If it does, we must apply the inverse of `b`, followed by traversal through `d`. We obtain the element by using the `inv_into` built-in function, which uses the Hilbert epsilon operator `SOME`. If not, the node remains on the right-hand side, requiring only the application of the `Inr` constructor.

```
fun s where
 "s (Inl e) = Inl (s_D e)"
|"s (Inr e) = (if e ∈ (E_R - b_E ‘ E_K) ∧ (s_R e ∈ b_V ‘ V_K)
  then Inl (d_V ((inv_into V_K b_V) (s_R e))) else Inr (s_R e))"
```

The target mapping follows analogous. However, both label mappings are straightforward and only require pattern matching with the respective labelling functions.

```
fun l where
  "l (Inl v) = l_D v"
| "l (Inr v) = l_R v"
```

Finally, we have completed defining all the necessary components for the `pre_graph` structure of a graph `H`. We use the `definition` keyword to define the final `pre_graph` structure `H`.

```
definition H where
<H ≡ ⦇nodes=V,edges=E
     ,source=s,target=t
     ,node_label=l,edge_label=m⦈>
```

To establish that this record represents a valid graph, we use the `sublocale` command, which necessitates us to discharge the required graph properties.

```
sublocale h: graph H
```

We proceed by defining the graph morphism $h\colon R \to H$, which is necessary if the node (or edge) resides in R without K via b. In the first scenario, it is evident that the node (or edge) is mapped through `Plus`, using the `Inr` constructor. In other instances, we utilize the inverse of b and proceed along d. In all cases, the node (or edge) invariably exists in the left part of `Plus`, indicated by the `Inl` constructor.

```
definition h where
<h ≡ ⦇node_map = λv. if v ∈ V_R - b_V ‘ V_K
                    then Inr v
                    else Inl (d_V ((inv_into V_K b_V) v)),
     edge_map = λe. if e ∈ E_R - b_E ‘ E_K
                    then Inr e
                    else Inl (d_E ((inv_into E_K b_E) e))⦈>
```

The morphism $c\colon D \to H$ is an injective morphism, which injects the graph D into H using the `Inl` constructor.

```
definition c :: "('e, 'e + 'g, 'f, 'f + 'h) pre_morph" where
  <c ≡ ⦇node_map = Inl, edge_map = Inl⦈>
```

In some cases, it is required to add additional type annotations. If these are skipped, Isabelle tries to infer the corresponding type, which might lead to a surprising outcome. The formalization of the deletion construction is described in [SP22].

2.5. Pushouts.

With these components in place, we demonstrate how they define the abstract concept of pushouts.

Definition 2.9 (Pushout)**.** Given graph morphisms $b\colon A \to B$ and $c\colon A \to C$, a graph D together with graph morphisms $f\colon B \to D$ and $g\colon C \to D$ is a *pushout* of $A \to B$ and $A \to C$ if the following holds (see Fig. 3):

(1) Commutativity: $f \circ b = g \circ c$, and
(2) Universal property: For all graph morphisms $p\colon B \to D'$ and $t\colon C \to D'$ such that $p \circ b = t \circ c$, there is a unique morphism $u\colon D \to D'$ such that $u \circ f = p$ and $u \circ g = t$.

We call D the *pushout object* and C the *pushout complement*. □

We are not able to express the unique existence, used in the *universal property* directly. The built-in binder for expressing this property presents a challenge. Since total functions are defined over the entire universe and our graphs may only cover a subset of the node (edge) identifiers, the built-in binder would be too strong. Therefore, our quantification needs to focus solely on the nodes (edges) pertinent to the required objects. As a result, we introduce the abbreviation `Ex1M`, which will quantify only over the required subset of nodes and edges used in the corresponding graphs.

$$\begin{tikzcd}
 & B \arrow[dr, "f"] \arrow[drr, bend left, "p"] & & \\
A \arrow[ur, "b"] \arrow[dr, "c"'] & & D \arrow[r, dashed] & D' \\
 & C \arrow[ur, "g"'] \arrow[urr, bend right, "t"'] & &
\end{tikzcd}$$

FIGURE 3. Pushout $A \to B \to D \leftarrow C \leftarrow A$

```
abbreviation Ex1M
  :: "(('v_1,'v_2,'e_1,'e_2) pre_morph ⇒ bool)
  ⇒ ('v_1,'e_1,'l,'m) pre_graph
  ⇒ bool"  where
  "Ex1M P E ≡ ∃x. P x ∧ (∀y. P y
                  ⟶ ( (∀e ∈ E_E. y_E e = x_E e)
                     ∧ (∀v ∈ V_E. y_V v = x_V v)))"
```

We are now prepared to formalize pushouts using a locale, addressing morphisms $A \to B$, $A \to C$, $B \to D$, and $C \to D$ that fulfil both commutativity and the universal property. The formalisation is given below:

```
locale pushout_diagram =
  b: morphism A B b +
  c: morphism A C c +
  f: morphism B D f +
  g: morphism C D g for A B C D b c f g +
assumes
  node_commutativity: <v ∈ V_A ⟹ f ∘→ b_V v = g ∘→ c_V v> and
  edge_commutativity: <e ∈ E_A ⟹ f ∘→ b_E e = g ∘→ c_E e> and
  universal_property: <⟦
    graph (D' :: ('c,'d) ngraph);
    morphism (to_ngraph B) D' x;
    morphism (to_ngraph C) D' y;
     ∀v ∈ V_to_ngraph A. x ∘→ (to_nmorph b)_V v = y ∘→ (to_nmorph c)_V v;
     ∀e ∈ E_to_ngraph A. x ∘→ (to_nmorph b)_E e = y ∘→ (to_nmorph c)_E e⟧
      ⟹ Ex1M (λu. morphism (to_ngraph D) D' u ∧
          (∀v ∈ V_to_ngraph B. u ∘→ (to_nmorph f)_V v = x_V v) ∧
          (∀e ∈ E_to_ngraph B. u ∘→ (to_nmorph f)_E e = x_E e) ∧
          (∀v ∈ V_to_ngraph C. u ∘→ (to_nmorph g)_V v = y_V v) ∧
          (∀e ∈ E_to_ngraph C. u ∘→ (to_nmorph g)_E e = y_E e))
          (to_ngraph D)>
```

The locale incorporates the four morphisms as depicted in Fig, 3. We express commutativity separately for nodes and edges through `node_commutativity` and `edge_commutativity`, respectively. Implementing the universal property proves more complex compared to using partial function. When using partial functions, we used equality between the two compositions while without, we explicitly have to quantify over the domain. We address the limitation of Isabelle's locale mechanism, which restricts quantification over fresh types used for the graph `D'`.

Our solution involves utilizing the `to_ngraph` and `from_ngraph` infrastructure to convert our morphism into natural numbers, a setup previously discussed. The need for this workaround arises from a limitation of simple type theories, which restricts higher-order logic (HOL) systems from allowing explicit quantification over polymorphic type variables [HUW14]. Consequently, certain concepts cannot be expressed directly, leading to details that may be hard to read and differ from the presentation in standard mathematical textbooks.

Furthermore, our reliance on total functions means we cannot verify the equality of morphisms directly. Instead, we must quantify over the specific areas they are defined in.

To respond to the limitations of locales, we have introduced an additional lemma within the locale's context. This lemma generalizes the universal property by omitting the `ngraph` topic. Consequently, we can depend on this lemma for most of our work, eliminating the need to consider the conversion between identifiers in `graph` and `ngraph`.

```
lemma universal_property_exist_gen:
  fixes D'
  assumes ‹graph D'› ‹morphism B D' x› ‹morphism C D' y›
    ‹∀v ∈ V_A. x ∘→ b_V v = y ∘→ c_V v›
    ‹∀e ∈ E_A. x ∘→ b_E e = y ∘→ c_E e›
  shows ‹Ex1M (λu. morphism D D' u ∧
            (∀v ∈ V_B. u ∘→ f_V v = x_V v) ∧
            (∀e ∈ E_B. u ∘→ f_E e = x_E e) ∧
            (∀v ∈ V_C. u ∘→ g_V v = y_V v) ∧
            (∀e ∈ E_C. u ∘→ g_E e = y_E e)) D›
```

The proof relies on the correctness property of the conversion functions.

Lemma 2.10 (Correctness of `to_ngraph`). *Let G be a graph, then `to_ngraph G` is a natural graph.*

The correctness property is expressed in Isabelle/HOL as an if and only if.

```
lemma graph_ngraph_corres_iff:
  ‹graph (to_ngraph G) ⟷ graph G ›
```

We express a similar property for morphisms and the lifted morphisms over to natural graphs.

Lemma 2.11 (Correctness of `to_nmorph`). *Let $m\colon G \to H$ be a graph morphism, then `to_nmorph m` is a morphism `to_ngraph G` to `to_ngraph H`.*

The corresponding formalisation is given by:

```
lemma morph_eq_nmorph_iff:
  ‹morphism G H m ⟷ morphism (to_ngraph G) (to_ngraph H) (to_nmorph m)›
```

We easily discharge both lemmas by unfolding the definitions of `to_ngraph` and `to_nmorph`, leveraging the injectivity of the corresponding functions and Isabelle's powerful automation capabilities.

An important property is that pushouts are unique up to isomorphism, which we have initially formalised in [SP22] and in this paper enhanced using the previously described techniques.

Theorem 2.12 (Uniqueness of pushouts [EEPT06]). *Let $b\colon A \to B$ and $c\colon A \to C$ together with D induce a pushout as depicted in Fig. 3. A graph D' together with morphisms $p\colon B \to D'$ and $t\colon C \to H$ is a pushout of b and c if and only if there is an isomorphism $u\colon D \to D'$ such that $u \circ f = p$ and $u \circ g = t$.* □

We formalise this property by entering the locale `pushout_diagram` context, which incorporates all relevant locale assumptions for use with their specified names. As a result, we are able to assume only the additional graph D' together with $B \to D'$ and $C \to D'$.

```
theorem uniqueness_po:
  fixes D'
  assumes
    D': <graph D'> and
    f': <morphism B D' f'> and
    g': <morphism C D' g'>
  shows <pushout_diagram  A B C D' b c f' g'
   ⟷ (∃u. bijective_morphism D D' u
        ∧ (∀v ∈ V_B. u ∘→ f_V v = f'_V v) ∧ (∀e ∈ E_B. u ∘→ f_E e = f'_E e)
        ∧ (∀v ∈ V_C. u ∘→ g_V v = g'_V v) ∧ (∀e ∈ E_C. u ∘→ g_E e = g'_E e))>
```

In our particular case, where we have injective pushouts (and rules), we can also state the uniqueness of the pushout complement.

Theorem 2.13 (Uniqueness of the pushout complement)**.** *Let* $b\colon A \to B$ *and* $c\colon A \to C$ *be injective morphisms, and the graph* D *induce a pushout as depicted in Fig. 3. Let* $A \to B \to D \leftarrow C' \leftarrow A$ *be another pushout with injective* $A \to C'$*. Then* C *and* C' *are isomorphic.* □

We formalise this theorem inside the context of the `pushout_diagram` locale. Since the local, in general, relies on morphisms, we need to assume, $A \to B$ and $A \to C$ are injective. From our assumption, that $C' \to D$ is a valid morphism, we could conclude that C' is a valid graph. Here, we explicitly added the assumption for clarity.

```
theorem uniqueness_pc:
  fixes C' c' g'
  assumes
    b: <injective_morphism A B b> and
    c: <injective_morphism A C c> and
    C': <graph C'> and
    c': <injective_morphism A C' c'> and
    g': <morphism C' D g'>
  shows <pushout_diagram A B C' D b c' f g'
        ⟶ (∃u. bijective_morphism C C' u)>
```

We discuss the proof in Section 3.

2.6. Direct Derivations.

The transformation of graphs by rules gives rise to *direct derivations*. The so-called dangling condition is a crucial requirement that ensures the resulting graph remains well-formed after applying a rule. It prevents the creation of *dangling* edges, which are edges that would be left pointing to non-existent nodes after the deletion of nodes. In essence, the dangling condition guarantees that when a node is deleted, all edges connected to it must either be explicitly deleted by the rule or be preserved through the interface graph. We describe the formalisation in [SP22].

Definition 2.14 (Direct derivation)**.** Let G and H be graphs, $r = \langle L \leftarrow K \to R\rangle$ be a rule, and $g\colon L \to G$ an injective morphism satisfying the dangling condition. Then G *directly derives* H by r and g, denoted by $G \Rightarrow_{r,g} H$, as depicted in the double pushout diagram in Fig. 4. □

$$\begin{tikzcd}
L & K \arrow[l] \arrow[r] \arrow[d] \arrow[dl, phantom, "(1)"] \arrow[dr, phantom, "(2)"] & R \arrow[d] \\
G \arrow[from=u, "g"'] & D \arrow[l] \arrow[r] & H
\end{tikzcd}$$

Figure 4. Double-pushout diagram

$$\begin{tikzcd}
& & B \arrow[dr, "f"] & \\
A' \arrow[r, dashed] \arrow[urr, bend left, "p"] \arrow[drr, bend right, "t"'] & A \arrow[ur, "b"] \arrow[dr, "c"'] & & D \\
& & C \arrow[ur, "g"'] &
\end{tikzcd}$$

Figure 5. Pullback $D \leftarrow C \leftarrow A \rightarrow B \rightarrow D$

Note that the injectivity of the matching morphism $g\colon L \rightarrow G$ leads to a DPO approach that is more expressive than in the case of arbitrary matches [HMP01]. In our earlier work [SP22], we used the `direct_derivation` locale to represent the operational view using gluing and deletion. Instead, here we use the categorical definition relying on pushouts.

Our formalisation is close to the Def. 2.14. We inherit the `rule` locale, together with an injective morphism g and the two pushouts (1) and (2). Since we introduced the `pre_rule` record, we use the accessor functions `lhs`, `rhs`, and `interf` to extract the corresponding objects.

```
locale direct_derivation =
  r: rule r b b' +
  gi: injective_morphism "lhs r" G g  +
  po1: pushout_diagram "interf r" "lhs r" D G b d g c +
  po2: pushout_diagram "interf r" "rhs r" D H b' d f c'
  for r b b' G g D d c H f c'
```

The operational definition is available within the `direct_derivation_construction` locale.

```
locale direct_derivation_construction =
  r: rule r b b' +
  d: deletion "interf r" G "lhs r" g b +
  g: gluing "interf r" d.D "rhs r" d.d b' for G r b b' g H +
assumes a: <H = g.H>
```

Here, we follow the same pattern but instead of the `pushout_diagram` locale, we rely on `deletion` and `gluing`. Inside the `for` statement, we introduce a fresh `pre_graph` record, named H, which represents the final graph, depicted in Fig. 4. By using the `assumes` statement, we require that the pushout object from (2) is equal to H (`H = g.H`). This allows to use the bound name H, instead of `g.H`, inside the proof.

2.7. **Pullbacks.** A pullback is dual to the concept of pushout and Pullbacks generally represent the intersection of objects over a common object.

Definition 2.15 (Pullback)**.** Given graph morphism $f\colon B \to D$ and $g\colon C \to D$, a graph A together with graph morphisms $b\colon A \to B$ and $c\colon A \to C$ is a *pullback* of $C \to D \leftarrow B$ if the following holds (see Fig. 5):

(1) Commutativity: $f \circ b = g \circ c$, and
(2) Universal property: For all graph morphisms $p\colon A' \to B$ and $t\colon A' \to C$ such that $f \circ p = g \circ t$, there is a unique morphism $u\colon H \to A$ such that $b \circ u = p$ and $c \circ u = t$. □

The formalisation closely follows the `pushout_diagram` locale, and all design considerations also apply here.

```
locale pullback_diagram =
  b: morphism A B b +
  c: morphism A C c +
  f: morphism B D f +
  g: morphism C D g for A B C D b c f g +
assumes
  node_commutativity: <⋀v. v ∈ V_A ⟹ f ∘→ b_V v = g ∘→ c_V v> and
  edge_commutativity: <⋀e. e ∈ E_A ⟹ f ∘→ b_E e = g ∘→ c_E e> and
  universal_property: <⟦
    graph (A' :: ('c,'d) ngraph);
    morphism A' C c';
    morphism A' B b';
     ⋀v. v ∈ V_A' ⟹ f ∘→ b'_V v = g ∘→ c'_V v;
     ⋀e. e ∈ E_A' ⟹ f ∘→ b'_E e = g ∘→ c'_E e⟧
    ⟹ Ex1M (λu. morphism A' A u ∧
             (∀v ∈ V_A'. b ∘→ u_V v = b'_V v) ∧
             (∀e ∈ E_A'. b ∘→ u_E e = b'_E e) ∧
             (∀v ∈ V_A'. c ∘→ u_V v = c'_V v) ∧
             (∀e ∈ E_A'. c ∘→ u_E e = c'_E e))
             A'>
```

Many of our proofs use the set-based construction of pullbacks, which is given by the following definition.

Definition 2.16 (Pullback construction [EEPT06])**.** Let $f\colon B \to D$ and $g\colon C \to D$ be graph morphisms. Then the following defines a graph A (see Fig. 5), the pullback object of f and g:

(1) $A = \{\langle x, y\rangle \in B \times C \mid f(x) = g(y)\}$ for nodes and edges, respectively
(2) $s_A(\langle x, y\rangle) = \langle s_B(x), s_c(y)\rangle$ for $\langle x, y\rangle \in E_B \times E_C$
(3) $t_A(\langle x, y\rangle) = \langle t_B(x), t_c(y)\rangle$ for $\langle x, y\rangle \in E_B \times E_C$
(4) $l_A(\langle x, y\rangle) = l_B(x)$ for $\langle x, y\rangle \in V_B \times V_C$
(5) $m_A(\langle x, y\rangle) = m_B(x)$ for $\langle x, y\rangle \in E_B \times E_C$
(6) $b\colon A \to B$ and $c\colon A \to C$ are defined by $b(\langle x, y\rangle) = x$ and $c(\langle x, y\rangle) = y$ □

We formalise the pullback construction using the `pullback_construction` locale, assuming the graph morphisms $f\colon B \to D$ and $g\colon C \to D$.

```
locale pullback_construction =
  f: morphism B D f +
  g: morphism C D g
  for B D C f g
```

To construct the pullback object C, we define all the required `pre_graph` components individually, starting with the node and edge sets as follows:

```
abbreviation V where
  ‹V ≡ {(x,y). x ∈ V_B ∧ y ∈ V_C ∧ f_V x = g_V y}›

abbreviation E where
  ‹E ≡ {(x,y). x ∈ E_B ∧ y ∈ E_C ∧ f_E x = g_E y}›
```

The corresponding set contains the pairs (x, y) where $x \in B$ and $y \in C$ such that they project via f and g to the same element in D. Both source and target functions (`s` and `t`), use the corresponding function on each element individually. For the labelling functions, it does not matter and we picked to use the one from B.

```
fun s where ‹s (x,y) = (s_B x, s_C y)›
fun t where ‹t (x,y) = (t_B x, t_C y)›
fun l where ‹l (x,_) = l_B x›
fun m where ‹m (x,_) = m_B x›
```

With all components available, we can now define the final `pre_graph` object:

```
definition A where
  ‹A ≡ ⦇nodes = V, edges = E, source = s, target = t
       ,node_label = l, edge_label = m⦈›
```

In subsequent code, we used the `sublocale` keyword to show that our definition of the pullback object A is a valid graph. We continue by defining the two missing morphisms we need to show the pullback diagram locale. First, the morphism $A \to B$ is defined by using the first element of the tuple. We archive this by using the built-in function `fst`. Note, we explicitly added type annotation to both `pre_morph` objects to prevent later problems if the type is inferred to strict.

```
definition b :: "('a × 'g, 'a, 'b × 'h, 'b) pre_morph" where
  ‹b ≡ ⦇node_map = fst, edge_map = fst⦈›
```

For $A \to C$, we use the other projection, `snd`, to extract the second element of the tuple.

```
definition c :: "('a × 'g, 'g, 'b × 'h, 'h) pre_morph"
  where ‹c ≡ ⦇node_map = snd, edge_map = snd⦈›
```

Finally, we prove that both, `b` and `c` definitions, are valid morphisms by using the `sublocale` mechanism. The proof of $b\colon A \to B$ follows by unfolding the definition of A and b while $c\colon A \to D$ also needs to account for the preservation of labels of f and g.

The next lemma shows that this construction leads to a valid pullback diagram.

Lemma 2.17 (Correctness of pullback construction)**.** *Let $f\colon B \to D$ and $g\colon C \to D$ be graph morphisms and let graph A and graph morphisms b and c be defined as in Def. 2.16. Then the square in Fig. 5 is a pullback diagram.*

We use the `sublocale` command, instead of `interpretation`, to make these facts persistent in the current context via the `pb` identifier.

```
sublocale pb: pullback_diagram A B C D b c f g
```

The proof basically follows from our construction. Similar to pushouts, pullbacks are unique up to isomorphism.

Theorem 2.18 (Uniqueness of pullbacks)**.** *Let $f\colon B \to D$ and $g\colon C \to D$ together with A induce a pullback as depicted in Fig. 5. A graph A' together with morphisms $p\colon A' \to B$ and $t\colon A' \to C$ is a pullback of f and g if and only if there is an isomorphism $u\colon A' \to A$ such that $b \circ u = p$ and $c \circ u = t$.*

$$\begin{tikzcd}
A \arrow[r] \arrow[d] \arrow[rd, phantom, "(1)"] & B \arrow[r] \arrow[d] \arrow[rd, phantom, "(2)"] & E \arrow[d] \arrow[rdd] & \\
C \arrow[r] \arrow[rrrd] & D \arrow[r] \arrow[rrd] & F & \\
 & & & X
\end{tikzcd}$$

Figure 6. Composite commutative diagram

The theorem is stated as follows in Isabelle/HOL:

```
theorem uniqueness_pb:
  fixes A' b' c'
  assumes
    A': <graph A'> and
    b': <morphism A' B b'> and
    c': <morphism A' C c'>
  shows <pullback_diagram A' B C D b' c' f g
          ⟷ (∃u. bijective_morphism A' A u
                ∧ (∀v ∈ V_A'. b ∘→ u_V v = b'_V v)
                ∧ (∀e ∈ E_A'. b ∘→ u_E e = b'_E e)
                ∧ (∀v ∈ V_A'. c ∘→ u_V v = c'_V v)
                ∧ (∀e ∈ E_A'. c ∘→ u_E e = c'_E e))>
```

The proof is dual to the uniqueness of the pushout (cf. Theorem 2.12) and can be found in [EEPT06].

2.8. **Composition and Decomposition of Pullbacks and Pushouts.** Essential properties for the forthcoming proofs in Section 3 and Section 4 are the composition and decomposition of pushouts and pullbacks.

Lemma 2.19 (Pushout/Pullback composition and decomposition)**.** *Given the commutative diagram in Fig. 6, then the following statements are true:*

(a) *If* (1) *and* (2) *are pushouts, so is* (1) + (2)
(b) *If* (1) *and* (1) + (2) *are pushouts, so is* (2)
(c) *If* (1) *and* (2) *are pullbacks, so is* (1) + (2)
(d) *If* (2) *and* (1) + (2) *are pullbacks, so is* (1) □

Proof. The following proofs are based on [EEPT06]:

(a) Assume (1) and (2) are pushouts. By using Lemma 2.6, the compositions $A \to B \to E$ and $C \to D \to F$ are morphisms. Let us first show commutativity:

$$\begin{aligned} A \to B \to E \to F &= A \to B \to D \to F && \text{(by commutativity of (2))} \\ &= A \to C \to D \to F && \text{(by commutativity of (1))} \end{aligned}$$

Finally, we show the universal property: Let X be a graph, and $E \to X$ and $C \to X$ be morphisms, such that $A \to B \to E \to X = A \to C \to X$. By the universal property of (1) using $B \to E \to X$ and $C \to X$, we obtain the unique morphism $D \to X$. By the universal property of (2) using $D \to X$ and $E \to X$, we obtain the unique morphism $F \to X$. Now we first show that $E \to F \to X = E \to X$ and

$C \to D \to F \to X = C \to X$. The first equation is true by the construction of $F \to X$. The second equation follows by:

$$\begin{aligned} C \to D \to F \to X &= C \to D \to X && \text{(by construction of } F \to X\text{)} \\ &= C \to X && \text{(by construction of } D \to X\text{)} \end{aligned}$$

By the universal property of (2), we know $F \to X$ is unique such that $E \to F \to X = E \to X$ and $D \to F \to X = D \to X$. We compose the second equation with the morphism $C \to D$ on both sides, so we get $C \to D \to F \to X = C \to D \to X = C \to X$ by construction of $D \to X$. Hence $F \to X$ is unique such that $E \to F \to X = E \to X$ and $C \to D \to F \to X = C \to X$.

(b) Assume (1) and (1) + (2) are pushouts and let X be a graph together with morphisms $D \to X$ and $E \to X$ such that $B \to E \to X = B \to D \to X$. Let $F \to X$ be the unique morphism resulting from the universal property of (1) + (2) such that $E \to F \to X = E \to X$ and

$$C \to D \to F \to X = C \to X.$$

But we also know that from the universal property of (1), that $D \to X$ is unique such that

$$C \to D \to X = C \to X.$$

Hence $D \to F \to X = D \to X$ because of the uniqueness of $D \to X$.

(c) Proof is analogous via dualisation.

(d) Proof is analogous via dualisation. □

Our Isabelle formalisation is an adaptation of a given proof, incorporating necessary technical modifications while maintaining the essence of the original argument. The pushout composition is a standalone lemma defined in the theory `Pushout`, cf. Fig. 1, following (a) of Lemma 2.19.

```
lemma pushout_composition:
  assumes
    1: <pushout_diagram A B C D f g g’ f’> and
    2: <pushout_diagram B E D F e g’ e’’ e’>
  shows <pushout_diagram A E C F (e ∘→ f) g  e’’ (e’ ∘→ f’)>
```

Here, $f\colon A \to B$, $g\colon A \to C$, $g'\colon B \to D$ and $f'\colon C \to D$ together with $e\colon B \to E$, $e'\colon D \to F$, and $e''\colon E \to F$ forms the composed pushout diagram.

For the pushout decomposition, we additionally assume that (2) commutes, i.e., $e'' \circ e = e' \circ g'$. In Isabelle, we express this independently for the nodes and edge of B.

```
lemma pushout_decomposition:
  assumes
    e : <morphism B E e> and
    e’: <morphism D F e’> and
    1 : <pushout_diagram A B C D f g g’ f’> and
    "1+2": <pushout_diagram A E C F (e ∘→ f) g  e’’ (e’ ∘→ f’)> and
    "2cv": <⋀v. v ∈ V_B ⟹ e’’ ∘→ e_V v = e’ ∘→ g’_V v> and
    "2ce": <⋀ea. ea ∈ E_B ⟹ e’’ ∘→ e_E ea = e’ ∘→ g’_E ea>
  shows <pushout_diagram B E D F e g’ e’’ e’>
```

In our formalisation, the parts (c) and (d) of the pullback are developed analogously to the pushout ones.

$$\begin{tikzcd}
 & K \arrow[dl, "id"'] \arrow[dr, "h"] \arrow[ddd, "id"] & \\
K \arrow[dr, "d'"] \arrow[ddd, "b"'] & & U \arrow[dl, "k"] \arrow[ddd, "l"] \\
 & D' \arrow[ddd, "m"'] & \\
 & K \arrow[dl, "b"'] \arrow[dr, "d"] & \\
L \arrow[dr, "g"'] & & D \arrow[dl, "c"] \\
 & G &
\end{tikzcd}$$

Figure 7. Commutative cube based on direct derivation [LS04]

```
lemma pullback_composition:
  assumes
    1: ‹pullback_diagram A B C D f g g' f'› and
    2: ‹pullback_diagram B E D F e g' e'' e'›
  shows ‹pullback_diagram A E C F (e ∘→ f) g e'' (e' ∘→ f')›

lemma pullback_decomposition:
  assumes
    f:  ‹morphism A B f› and
    f': ‹morphism C D f'› and
    2: ‹pullback_diagram B E D F e g' e'' e'› and
    "1+2":  ‹pullback_diagram A E C F (e ∘→ f) g  e'' (e' ∘→ f')› and
    "1cv": ‹⋀v. v ∈ V_A ⟹ g' ∘→ f_V v = f' ∘→ g_V v› and
    "1ce": ‹⋀ea. ea ∈ E_A ⟹ g' ∘→ f_E ea = f' ∘→ g_E ea›
  shows ‹pullback_diagram A B C D f g g' f'›
```

The proof is analogous to [EEPT06, Fact 2.27]. In the next section, we show that direct derivations have a unique (up to isomorphism) result by making use of the infrastructure we have employed so far.

3. Uniqueness of Direct Derivations

The uniqueness of direct derivations is an important property when reasoning about rule applications. This section does not rely on the adhesiveness of the category of graphs, instead we base our proof on the characterisation of graph pushouts in [EK79]. Before stating the theorem, we introduce additional facts, mainly about pushouts and pullbacks, which are used within the proof of Theorem 3.9.

In general, pushouts along injective morphisms are also pullbacks.

Lemma 3.1 (Injective pushouts are pullbacks [EEPT06])**.** *A pushout diagram as depicted in Fig. 3 is also a pullback if $A \to B$ and $A \to C$ are injective.*

The proof relies on the pullback construction (cf. Def. 2.16) and the fact that pullbacks are unique (cf. Theorem 2.18). We show this property in our formalisation within the `Gluing` theory (cf. Fig. 1).

```
lemma pushout_pullback_inj_b:
  assumes
    b: ‹injective_morphism A B b› and
    c: ‹injective_morphism A C c›
  shows ‹pullback_diagram A B C D b c f g›
```

Furthermore, pushouts and pullbacks preserve injectivity (surjectivity) in the sense that the opposite morphism of the corresponding diagram (see Fig. 3 and Fig. 5) is also injective (surjective).

Lemma 3.2 (Preservation of injective and surjective morphisms [EEPT06]). *Given a pushout diagram in Fig. 3, if $A \to B$ is injective (surjective), so is $C \to D$. Given a pullback diagram in Fig. 5, if $C \to D$ is injective (surjective), so is $A \to B$.*

We formalise these properties independent of the pushout, pullback and injectivity or surjectivity using our infrastructure. Inside the `pushout` locale context, if $b\colon A \to B$ is injective, so is $g\colon C \to D$ as depicted in Fig. 3.

```
lemma b_inj_imp_g_inj:
  assumes ‹injective_morphism A B b›
  shows ‹injective_morphism C D g›
```

In case, $b\colon A \to B$ is surjective, so is $g\colon C \to D$.

```
lemma b_surj_imp_g_surj:
  assumes ‹surjective_morphism A B b›
  shows ‹surjective_morphism C D g›
```

Consequently, in case $A \to B$ is bijective, so is $C \to D$.

```
lemma b_bij_imp_g_bij:
  assumes ‹bijective_morphism A B b›
  shows ‹bijective_morphism C D g›
```

The pullback statements are done similar in the corresponding locale context.

Certain forms of commutative diagrams give rise to pullbacks. This property is used in the proof of the uniqueness of the pushout complement (cf. Theorem 3.9).

Lemma 3.3 (Special pullbacks [EEPT06]). *The commutative diagram in Fig. 9 is a pullback if m is injective.*

In Isabelle, we describe this lemma as follows.

```
lemma fun_algrtr_4_7_2:
  fixes C A m
  assumes ‹injective_morphism C A m›
  shows ‹pullback_diagram C C C A idM idM m m›
```

Isabelle is able to discharge this goal using supplied facts automatically.

Definition 3.4 (Reduced chain-condition [EK79]). The commutative diagram in Fig. 8 satisfies the *reduced chain-condition*, if for all $b' \in B$ and $c' \in C$ with $f(b') = g(c')$ there is $a \in A$ such that $b(a) = b'$ and $c(a) = c'$.

We show that pullbacks satisfy the reduced chain-condition.

Lemma 3.5 (Pullbacks satisfy the reduced chain-condition). *Each pullback diagram as depicted in Fig. 5 satisfies the reduced chain-condition.*

We first prove this lemma, separately for nodes and edges, state this lemma in Isabelle as follows:

```
lemma reduced_chain_condition_nodes:
  fixes x y
  assumes <x ∈ V_B> <y ∈ V_C> <f_V x = g_V y>
  shows <∃a ∈ V_A. (b_V a = x ∧ c_V a = y)>
```

Isabelle's simplifier is able to discharge all goals if supplied with the corresponding facts. Especially, that the construction is also a pullback (commutativity) and the definition of the graph A, together with the two morphisms. Subsequently, we can state a more generalised lemma inside the `pullback_diagram` locale:

```
lemma (in pullback_diagram) reduced_chain_condition_nodes:
  fixes x y
  assumes <x ∈ V_B> <y ∈ V_C> <f_V x = g_V y>
  shows <∃a ∈ V_A. (b_V a = x ∧ c_V a = y)>
```

Our proof relies on the pullback construction (cf. Def. 2.16) and the fact that pullbacks are unique (cf. Theorem 2.18).

Definition 3.6 (Jointly surjective)**.** Given injective graph morphisms $f\colon B \to D$ and $g\colon C \to D$. f and g are *jointly surjective*, if each item in D has a preimage in B or C.

Lemma 3.7 (Pushouts are jointly surjective)**.** *Given the pushout diagram in Fig. 3. The pair* (f, g) *is* jointly surjective.

We express this property for edges in Isabelle as follows:

```
lemma joint_surjectivity_edges:
  fixes x
  assumes <x ∈ E_D>
  shows <(∃e ∈ E_C. g_E e = x) ∨ (∃e ∈ E_B. f_E e = x)>
```

The proof is by contradiction. We construct the graph D' using the disjoint union with an edge $e \in D$, such that it has no preimage in B or C via f and g. We can now construct two different morphisms from D' to D, which contradicts the uniqueness property of the pushout.

Theorem 3.8 (Pushout characterization [EK79])**.** *The commutative diagram in Fig. 8 is a pushout, if the following conditions are true:*

(1) *The morphisms b, c, f, g are injective.*
(2) *The diagram satisfies the reduced chain-condition.*
(3) *The morphisms g, f are jointly surjective.*

$$\begin{tikzcd}
& A \arrow[dl, "c"'] \arrow[dr, "b"] & \\
C \arrow[dr, "g"'] & & B \arrow[dl, "f"] \\
& D &
\end{tikzcd}$$

Figure 8. Commutative diagram

$$\begin{tikzcd}
& & C \arrow[dr, "m"] & \\
H \arrow[r, dashed] \arrow[urr, bend left] \arrow[drr, bend right] & C \arrow[ur, "id"] \arrow[dr, "id"'] & & D \\
& & C \arrow[ur, "m"'] &
\end{tikzcd}$$

Figure 9. Special pullback diagram

```
lemma po_characterization:
  assumes
    b: ‹injective_morphism A B b› and
    c: ‹injective_morphism A C c› and
    f: ‹injective_morphism B D f› and
    g: ‹injective_morphism C D g› and
    node_commutativity: ‹⋀v. v ∈ V_A ⟹ f ∘→ bV v = g ∘→ cV v› and
    edge_commutativity: ‹⋀e. e ∈ E_A ⟹ f ∘→ bE e = g ∘→ cE e› and
    reduced_chain_condition_nodes:
      ‹⋀x y. x ∈ V_B ⟹ y ∈ V_C ⟹ fV x = gV y
                          ⟹ (∃a ∈ V_A. (bV a = x ∧ cV a = y))› and
    reduced_chain_condition_edges:
      ‹⋀x y. x ∈ E_B ⟹ y ∈ E_C ⟹ fE x = gE y
                          ⟹ (∃a ∈ E_A. (bE a = x ∧ cE a = y))› and
    joint_surjectivity_nodes:
          ‹⋀x. x ∈ V_D ⟹ (∃v ∈ V_C. gV v = x) ∨ (∃v ∈ V_B. fV v = x)› and
    joint_surjectivity_edges:
          ‹⋀x. x ∈ E_D ⟹ (∃e ∈ E_C. gE e = x) ∨ (∃e ∈ E_B. fE e = x)›
  shows ‹pushout_diagram A B C D b c f g›
```

The following theorem implies the uniqueness of the pushout complement, which is known to hold if the morphism $K \to L$ in the applied rule is injective, even if the matching morphism $L \to G$ is non-injective [Ros75]. In our case, both morphisms are injective.

Theorem 3.9 (Uniqueness of direct derivations)**.** *Let* (1) + (2) *and* (3) + (4) *be direct derivations as depicted in Fig. 12. Then* $D \cong D'$ *and* $H \cong H'$.

This theorem is stated in Isabelle/HOL within the locale `direct_derivation`. Within the `assumes` part, the second direct derivation, which we call `dd2`, is introduced.

```
theorem uniqueness_direct_derivation:
  assumes
    dd2: ‹direct_derivation r b b' G g D' d' m H' f' m'›
  shows ‹(∃u. bijective_morphism D D' u)
         ∧ (∃u. bijective_morphism H H' u)›
```

The uniqueness proof of direct derivations (see Fig.12) is performed in two phases. Firstly, we show the uniqueness of the pushout complement, which was first shown by Rosen [Ros75]. Subsequently, we show that given a bijection between D and D', the pushout object is also unique up to isomorphism.

Theorem 3.9. The first phase of our proof closely follows Lack and Sobocinski [LS04], except for the final step, where the authors rely on adhesiveness. We finish the proof by relying on the pushout characterization (cf. Theorem 3.8). Given the two pushout diagrams (1) and (3) in Fig. 12 with injective $K \to D$ and $K \to D'$. To show the existence of a bijection between D and D', we construct the commutative cube in Fig. 7 with (1) as the bottom face and (3) as the front-left face, and show that l and k are bijections. For the latter, we show that the back-right and top faces are pushouts. (In [LS04], this is shown by adhesiveness, while we argue with the pushout characterization of Theorem 3.8.) The front-right face is a pullback construction (cf. Def. 2.16) which we tell Isabelle by interpretation of the `pullback_construction` locale.

```
interpret fr: pullback_construction D G D' c m ..
```

We use Isabelle's shorthand notation `..` for the `standard` tactic, to discharge the proof obligations which follow from the assumptions. Note that the pullback object together with the two morphisms is specified within the locale. Subsequent code will reference the pullback object by `fr.A`, the morphism l by `fr.b` and k by `fr.c` (see Fig. 7). (The identifiers within locales are given by the definition. As a result, the pullback object of the front-right face is referred to as `A` rather than the interpretation parameter `K`.) From Lemma 3.3, in our formalisation referenced by `fun_algrtr_4_7_2`, we know that the back-left face is a pullback.

```
interpret bl: pullback_diagram "interf r" "interf r"
                "interf r" "lhs r" idM idM b b
  using fun_algrtr_4_7_2[OF r.k.injective_morphism_axioms]
  by assumption
```

To show that the back-right face is a pullback, we start with the front-left face. As the front left face is a pushout and m is injective, n' is too (cf. Lemma 3.2). Since pushouts along injective morphisms are also pullbacks (cf. Lemma 3.1), the front-left face is also a pullback. Using the pullback composition (cf. Lemma 2.19), the back face is pullback.

```
interpret backside: pullback_diagram "interf r" D' "interf r" G
                     ‹d' ∘→ idM› idM m ‹g ∘→ b›
  using pullback_composition[OF bl.pullback_diagram_axioms
                                 dd2.pb1.flip_diagram]
  by assumption
```

We define $h\colon K \to U$ using both, the d and d' morphisms as $h\ x = (d\ x, d'\ x)$ and subsequently prove the morphism properties.

```
define h where
  ‹h ≡ ⦇node_map = λv. (dV v, d'V v)
       ,edge_map = λe. (dE e, d'E e)⦈›
```

We follow by showing that the top and bottom face commutes, i.e., $d' \circ id = k \circ h$ and $g \circ b = c \circ d$, respectively. This establishes the fact, that the right-side of the cube is a pullback. Using the pullback decomposition (cf. Lemma 2.19), the back-right face is a pullback. To approach the top-face, we start by showing it is a pullback, and subsequently it is also a pushout. Since m is injective, from Lemma 3.1 we know the bottom-face is also a pullback. Using the pullback composition (cf. Lemma 2.19), the bottom and back-left face is a pullback. By commutativity of the bottom face $g \circ b = c \circ d$ and back-right face $l \circ h = d \circ id$, the front-right and top face is a pullback. By the pullback decomposition (cf. Lemma 2.19) we can show that the top face is a pullback. We show this pullback is also a pushout by using the pushout characterization (cf. Theorem 3.8). Therefore, we need to show that h is injective, which follows from the construction above of h.

```
interpret h: injective_morphism "interf r" fr.A h
proof
  show <inj_on hV Vinterf r>
    using d_inj.inj_nodes
    by (simp add: h_def inj_on_def)
next
  show <inj_on hE Einterf r>
    using d_inj.inj_edges
    by (simp add: h_def inj_on_def)
qed
```

Joint surjectivity of k and d' follows from the pullback construction and the reduced-chain condition (cf. Lemma 3.4) of the front-left and top face. Note, that the reduced-chain condition holds for all pullbacks. Finally, we need to show that k and l are bijections. Since the top face is a pushout and the $C \to C$ morphism is a bijection, by Lemma 3.2, so is k.

```
interpret k_bij: bijective_morphism fr.A D' fr.c
  using top.b_bij_imp_g_bij[OF r.k.G.idm.bijective_morphism_axioms]
  by assumption
```

To show l is a bijection, we show that the back-right face is a pushout by using the pushout characterization. The bijectivity of l follows from the fact pushouts preserve bijections. We follow by defining the morphism $u\colon D \to D'$ as the composition of the l^{-1} and k. The inverse of l is obtained by using a lemma within our formalisation of bijective morphisms, stating the existence of the inverse (`ex_inv`) using the `obtain` keyword:

```
obtain linv where linv:<bijective_morphism D fr.A linv>
  and <⋀v. v ∈ VD⟹ fr.b ∘→linvV v = v>
      <⋀e. e ∈ ED⟹ fr.b ∘→linvE e = e>
  and <⋀v. v ∈ Vfr.A⟹ linv ∘→ fr.b V v = v>
      <⋀e. e ∈ Efr.A⟹ linv ∘→ fr.b E e= e>
  by (metis l_bij.ex_inv)
```

We finish the first phase by defining the morphism $u\colon D \to D'$ as $k \circ l^{-1}$ and subsequently prove that morphism composition preserves bijections (using the already proven `bij_comp_is_bij` lemma).

```
define u where <u ≡ fr.c ∘→ linv>

interpret u: bijective_morphism D D' u
  using bij_comp_bij_is_bij[OF linv k_bij.bijective_morphism_axioms]
  by (simp add: u_def)
```

$$\begin{tikzcd}
& R \arrow[dr, "f"] \arrow[drr, bend left, "f'"] & & \\
K \arrow[ur, "b'"] \arrow[dr, "d"'] & & H \arrow[r, dashed, "u'"] & H' \\
& D \arrow[ur, "c'"] \arrow[urr, bend right, "m' \circ u"'] & &
\end{tikzcd}$$

FIGURE 10. Construction of u'

$$\begin{tikzcd}
& R \arrow[dr, "f'"] \arrow[drr, bend left, "f"] & & \\
K \arrow[ur, "b'"] \arrow[dr, "d'"'] & & H' \arrow[r, dashed, "u''"] & H \\
& D' \arrow[ur, "m'"] \arrow[urr, bend right, "c' \circ u^{-1}"'] & &
\end{tikzcd}$$

FIGURE 11. Construction of u''

The second phase is to show the existence of an isomorphism $H \to H'$. We start by obtaining $u' \colon H \to H'$ and $u'' \colon H' \to H$ and show they are inverses. We use the universal property of the pushout depicted in Fig. 10, which requires us to show commutativity: $f' \circ b' = m' \circ u \circ d$. So we substitute $u \circ d = d'$ into the commutativity equation of pushout (4) ($f' \circ b' = m' \circ d'$) in Fig. 12. We get $u \circ d = d'$ as follows: $u \circ d \overset{(1)}{=} k \circ l^{-1} \circ d \overset{(2)}{=} k \circ l^{-1} \circ l \circ h \overset{(3)}{=} k \circ h \overset{(4)}{=} d'$. Here, (1) is justified by the definition of u, (2) by the definition of l and h (which makes the back-right face in Fig. 7 commute), (3) by inverse cancellation, and finally (4) by the definitions of k and h (similarly to step (2)). We obtain $u'' \colon H' \to H$ by using the universal property of the pushout depicted in Fig. 11. We show the commutativity $f \circ b = c' \circ u^{-1} \circ d'$ by substituting $u^{-1} \circ d' = d$ into the commutativity equation of pushout (2) ($f \circ b' = c' \circ d$) in Fig. 12: $u^{-1} \circ d' \overset{(5)}{=} u^{-1} \circ u \circ d \overset{(6)}{=} d$. Here, (5) is justified by the above proven equation $u \circ d = d'$, and (6) follows from cancellation of inverses. The final steps are to show $u' \circ u'' = id$ and $u'' \circ u' = id$. To show the first equation, we start with $f' = u' \circ u'' \circ f'$ and $m' = u' \circ u'' \circ m'$, which we get from the definitions of u' and u''. Using the universal property of pushout (4) in Fig. 12 together with H', f', m', we conclude that the identity is the unique morphism $H' \to H'$ that makes the triangles commute. If $u' \circ u''$ makes the triangles commute as well, it is equal to the identity morphism. The first triangle commutes because $u' \circ u'' \circ f' \overset{(7)}{=} u' \circ f \overset{(8)}{=} f'$. Here, (7) and (8) are justified by the corresponding construction of u' and u'' (see the triangles in Fig. 10 and Fig. 11). For the second triangle, we start by using the commutativity of the bottom triangle in Fig. 10 and composing to the right with u^{-1}: $u' \circ c' \circ u^{-1} = m' \circ u \circ u^{-1}$. By cancellation of inverses we get $u' \circ c' \circ u^{-1} = m'$ and by substituting $c' \circ u^{-1}$ using the commutativity of the bottom triangle in Fig. 11, we prove that $u' \circ u'' \circ m' = m'$. Showing that $u'' \circ u' = id$ follows analogously and is omitted to save space. □

$$\begin{tikzcd}
L \arrow[d, "g"'] \arrow[dd, bend left, "g"] \arrow[dr, phantom, "(1)"] & K \arrow[l] \arrow[r] \arrow[d] \arrow[dd, bend left] \arrow[dr, phantom, "(2)"] & R \arrow[d] \arrow[dd, bend left] \\
G \arrow[dr, phantom, "(3)"] & D \arrow[l] \arrow[r] \arrow[dr, phantom, "(4)"] & H \\
G & D' \arrow[l] \arrow[r] & H'
\end{tikzcd}$$

Figure 12. Uniqueness of direct derivations

$$\begin{tikzcd}
 & G \arrow[dl, Rightarrow, "{p_1, m_1}"'] \arrow[dr, Rightarrow, "{p_2, m_2}"] & \\
H_1 \arrow[dr, Rightarrow, "{p_2, m_{2'}}"'] & & H_2 \arrow[dl, Rightarrow, "{p_1, m_{1'}}"] \\
 & G' &
\end{tikzcd}$$

Figure 13. Church-Rosser Theorem

With the uniqueness of direct derivations (cf. Theorem 3.9), we get the uniqueness of the pushout complement.

Corollary 3.10 (Uniqueness of pushout complements)**.** *Given a pushout as depicted in Fig. 3 where $A \to B$ is injective. Then the graph D is unique up to isomorphism.*

We omit the proof to conserve space. The upcoming section introduces the so-called Church-Rosser Theorem, which states that parallell independent direct derivations have the diamond property.

4. Church-Rosser Theorem

The Church-Rosser Theorem refers to the idea that two graph transformation rules can be applied independently of each other, either sequentially or in parallel, without changing the final result. We follow the independence characterization of direct derivations given in [EEPT06].

Definition 4.1 (Parallel independence [EEPT06])**.** The two direct derivations $G \Rightarrow_{p_1,m_1} H_1$ and $G \Rightarrow_{p_2,m_2} H_2$ in Fig. 14 are *parallel independent* if there exists morphisms $L_1 \to D_2$ and $L_2 \to D_1$ such that $L_1 \to D_2 \to G = L_1 \to G$ and $L_2 \to D_1 \to G = L_2 \to G$. □

$$\begin{tikzcd}
R_1 \arrow[d] \arrow[dr, phantom, "(2)"] & K_1 \arrow[l] \arrow[r] \arrow[d] \arrow[drr, phantom, "(1)"] & L_1 \arrow[dr, "m_1"'] \arrow[drrr, dashed] & & L_2 \arrow[dl, "m_2"] \arrow[dlll, dashed] & K_2 \arrow[l] \arrow[r] \arrow[d] \arrow[dll, phantom, "(3)"] \arrow[dr, phantom, "(4)"] & R_2 \arrow[d] \\
H_1 & D_1 \arrow[l] \arrow[rr] & & G & & D_2 \arrow[ll] \arrow[r] & H_2
\end{tikzcd}$$

FIGURE 14. Parallel independence

$$\begin{tikzcd}
L_1 \arrow[d] \arrow[dr, phantom, "(1)"] & K_1 \arrow[l] \arrow[r] \arrow[d] \arrow[drr, phantom, "(2)"] & R_1 \arrow[dr, "n_1"'] \arrow[drrr, dashed] & & L_2 \arrow[dl, "m_2"] \arrow[dlll, dashed] & K_2 \arrow[l] \arrow[r] \arrow[d] \arrow[dll, phantom, "(3)"] \arrow[dr, phantom, "(4)"] & R_2 \arrow[d] \\
G & D_1 \arrow[l] \arrow[rr] & & H & & D_2 \arrow[ll] \arrow[r] & G\prime
\end{tikzcd}$$

FIGURE 15. Sequential independence

```
locale parallel_independence =
  p₁: direct_derivation r₁ b₁ b₁' G g₁ D₁ m₁ c₁ H₁ f₁ h₁ +
  p₂: direct_derivation r₂ b₂ b₂' G g₂ D₂ m₂ c₂ H₂ f₂ h₂
  for r₁ b₁ b₁' G g₁ D₁ m₁ c₁ H₁ f₁ h₁
      r₂ b₂ b₂' g₂ D₂ m₂ c₂ H₂ f₂ h₂ +
  assumes
    i: ‹∃i. morphism (lhs r₁) D₂ i
           ∧ (∀v ∈ V_lhs r₁. c₂ ∘→ iV v = g₁V v)
           ∧ (∀e ∈ E_lhs r₁. c₂ ∘→ iE e = g₁E e)› and
    j: ‹∃j. morphism (lhs r₂) D₁ j
           ∧ (∀v ∈ V_lhs r₂. c₁ ∘→ jV v = g₂V v)
           ∧ (∀e ∈ E_lhs r₂. c₁ ∘→ jE e = g₂E e)›
```

Definition 4.2 (Sequential independence [EEPT06])**.** The two direct derivations $G \Rightarrow_{p_1,m_1} H_1$ and $G \Rightarrow_{p_2,m_2} H_2$ in Fig. 15 are *sequential independent* if there exists morphisms $R_1 \to D_2$ and $L_2 \to D_1$ such that $R_1 \to D_2 \to H = R_1 \to H$ and $L_2 \to D_1 \to H = L_2 \to H$. □

```
locale sequential_independence =
  p₁: direct_derivation r₁ b₁ b₁' G g₁ D₁ m₁ c₁ H₁ f₁ h₁ +
  p₂: direct_derivation r₂ b₂ b₂' H₁ g₂ D₂ m₂ c₂ H₂ f₂ h₂
  for r₁ b₁ b₁' G g₁ D₁ m₁ c₁ H₁ f₁ h₁
      r₂ b₂ b₂' g₂ D₂ m₂ c₂ H₂ f₂ h₂ +
  assumes
    i: ‹∃i. morphism (rhs r₁) D₂ i
           ∧ (∀v ∈ V_rhs r₁. c₂ ∘→ iV v = f₁V v)
           ∧ (∀e ∈ E_rhs r₁. c₂ ∘→ iE e = f₁E e)› and
    j: ‹∃j. morphism (lhs r₂) D₁ j
           ∧ (∀v ∈ V_lhs r₂. h₁ ∘→ jV v = g₂V v)
           ∧ (∀e ∈ E_lhs r₂. h₁ ∘→ jE e = g₂E e)›
```

Theorem 4.3 (Church-Rosser Theorem [EK76])**.** *Given two parallel independent direct derivations $G \Rightarrow_{p_1,m_1} H_1$ and $G \Rightarrow_{p_2,m_2} H_2$, there is a graph G' together with sequential independent direct derivations $H_1 \Rightarrow_{p_2,m'_2} G'$ and $H_2 \Rightarrow_{p_1,m'_1} G'$.*

Actually, we have shown more, namely that $G \Rightarrow_{p_1,m_1} H_1 \Rightarrow_{p_2,m'_2} G'$ and $G \Rightarrow_{p_2,m_2} H_2 \Rightarrow_{p_1,m'_1} G'$ are sequentially independent. We express this theorem in Isabelle/HOL within the `parallel_independence` locale as follows:

```
theorem (in parallel_independence) church_rosser:
  shows <∃g' D' m'c' H f' h' g'' D'' m'' c'' H f'' h''.
    sequential_independence r₁ b₁ b₁' G g₁ D₁ m₁ c₁ H₁ f₁ h₁ r₂
                            b₂ b₂' g' D' m' c' H f' h'
  ∧ sequential_independence r₂ b₂ b₂' G g₂ D₂ m₂ c₂ H₂ f₂ h₂ r₁
                            b₁ b₁' g'' D'' m'' c'' H f'' h''>
```

Theorem 4.3. We closely follow the original proof by Ehrig and Kreowksi [EK76] where in a first stage, the pushouts (1) - (4) of Fig. 14 are vertically decomposed into pushouts (11) + (11), (21) + (22), (31) + (32), and (41) + (42), as depicted in Fig. 16. In a second stage, these pushouts are rearranged as in Fig. 17 and the new pushout (5) is constructed. Subsequently, we prove the two vertical pushouts (11) and (12). The pushouts (31) and (32) follow analogously and are not shown to conserve space.

We start by constructing the pullback (12) which we bind to the symbol `c12`, allowing later references, using our `pullback_construction` locale.

```
interpret "c12": pullback_construction D₁ G D₂ c₁ c₂ ..
```

The existence of $K_1 \to D$ follows from the universal property, and $D \to D_2$ from the construction of the pullback (12):

```
obtain j₁ where <morphism (interf r₁) c12.A j₁>
  and <⋀v. v∈V_interf r₁ ⟹ c12.b ∘→ j₁V v = m₁V v>
      <⋀e. e∈E_interf r₁ ⟹ c12.b ∘→ j₁E e = m₁E e>
  and <⋀v. v∈V_interf r₁ ⟹ c12.c ∘→ j₁V v = i₁ ∘→ b₁V v>
      <⋀e. e∈E_interf r₁ ⟹ c12.c ∘→ j₁E e = i₁ ∘→ b₁E e>
  using c12.pb.universal_property_exist_gen[OF p₁.r.k.G.graph_axioms
            wf_b₁i₁.morphism_axioms p₁.po1.c.morphism_axioms a b]
  by fast
```

From the fact that (1) = (11) + (12), we know (11) + (12) is a pushout and since $K_1 \to L_1$ is injective, it is also a pullback (cf. Lemma 3.1). By pullback decomposition (cf. Lemma 2.19), (11) is a pullback. We use the pushout characterization (cf. Theorem 3.8) to show it is also a pushout, which requires us to show injectivity of all morphisms, reduced-chain condition, and joint surjectivity of $D \to D_2$ and $L_1 \to D_2$. The injectivity of $K_1 \to L_1$ is given, $D \to D_2$ follows from pushout (1) and the injectivity of $K_1 \to L_1$ (cf. Lemma 3.2). To show the injectivity of $L_1 \to D_2$, we use the parallel independence (cf. Def. 4.1) $L_1 \to D_2 \to G = L_1 \to G$ and the injectivity of $L_1 \to G$.To show injectivity of $K_1 \to D$, we use the triangle $L_1 \to D_2 \to G = L_1 \to G$ obtained by the universal property of pullback (12) and the injectivity of both, $L_1 \to G$ and $D_2 \to G$. The reduced-chain condition follows by Lemma 3.5. To show the joint surjectivity of $D \to D_2$ and $L_1 \to D_2$ (that is each x in D_2 has a preimage in either D or L_1). Let y be the image of x in G. We apply the joint surjectivity of pushout (11) + (12) to y, that is y has a preimage in either D_1 or L_1. In the former case (y has a preimage z in D_1): from the pullback construction (cf. Def. 2.16), we get the common preimage of z and x in D which shows the former case. In the latter case, y has a preimage in L_1 via D_2. Since $D_2 \to G$ is injective, that preimage is mapped via x which means, x has a preimage in L_1. This shows the latter case.

$$\begin{tikzcd}
R_1 \arrow[d] \arrow[dd, bend right] \arrow[dr, phantom, "(21)"] & K_1 \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "(11)"] & L_1 \arrow[d] \arrow[ddr, "m_1"] & & L_2 \arrow[d] \arrow[ddl, "m_2"'] \arrow[dr, phantom, "(31)"] & K_2 \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "(41)"] & R_2 \arrow[d] \arrow[dd, bend left] \\
\overline{D}_2 \arrow[d] \arrow[dr, phantom, "(22)"] & D \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "(12)"] & D_2 \arrow[dr] & & D_1 \arrow[dl] \arrow[dr, phantom, "(32)"] & D \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "(42)"] & \overline{D}_1 \arrow[d] \\
H_1 & D_1 \arrow[l] \arrow[rr] & & G & & D_2 \arrow[ll] \arrow[r] & H_2
\end{tikzcd}$$

FIGURE 16. Vertical pushout decomposition of Fig. 14

$$\begin{tikzcd}
L_1 \arrow[d] \arrow[dd, bend right, "m_1"'] \arrow[dr, phantom, "(11)"] & K_1 \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "(21)"] & R_1 \arrow[d] \arrow[ddr] & & L_2 \arrow[d] \arrow[ddl, "m_2'"'] \arrow[dr, phantom, "(31)"] & K_2 \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "(41)"] & R_2 \arrow[d] \arrow[dd, bend left] \\
D_2 \arrow[d] \arrow[dr, phantom, "(12)"] & D \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "(22)"] & \overline{D}_2 \arrow[dr] & & D_1 \arrow[dl] \arrow[dr, phantom, "(22)"] & D \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "(5)"] & \overline{D}_1 \arrow[d] \\
G & D_1 \arrow[l] \arrow[rr] & & H_1 & & \overline{D}_2 \arrow[ll] \arrow[r] & G'
\end{tikzcd}$$

$$\begin{tikzcd}
L_2 \arrow[d] \arrow[dd, bend right, "m_2"'] \arrow[dr, phantom, "(31)"] & K_2 \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "(41)"] & R_2 \arrow[d] \arrow[ddr] & & L_1 \arrow[d] \arrow[ddl, "m_1'"'] \arrow[dr, phantom, "(11)"] & K_1 \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "(21)"] & R_1 \arrow[d] \arrow[dd, bend left] \\
D_1 \arrow[d] \arrow[dr, phantom, "(32)"] & D \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "(42)"] & \overline{D}_1 \arrow[dr] & & D_2 \arrow[dl] \arrow[dr, phantom, "(42)"] & D \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "(5)"] & \overline{D}_2 \arrow[d] \\
G & D_2 \arrow[l] \arrow[rr] & & H_2 & & \overline{D}_1 \arrow[ll] \arrow[r] & G'
\end{tikzcd}$$

FIGURE 17. Rearranged pushouts of Fig. 16

The pushouts (21), (41) are constructed using the `gluing` locale (see [SP22] for a detailed description).

```
interpret "c21": gluing "interf r_1" c12.A "rhs r_1" j_1 b_1' ..
interpret "c41": gluing "interf r_2" c12.A "rhs r_2" j_2 b_2' ..
```

The existence of $\overline{D}_2 \to H_1$ and $\overline{D}_1 \to H_2$ follows from the universal property of pushout (21) and (41), respectively. Pushouts (22) and (42) are obtained using the pushout decomposition (see Lemma 2.19). This finishes the first stage of the proof. The second stage rearranges the pushouts, as depicted in Fig. 17, such that we obtain two direct derivations $H_1 \Rightarrow_{p_2,m_2'} G'$ and $H_2 \Rightarrow_{p_1,m_1'} G'$. Here, we compose the pushouts from stage 1 (see Lemma 2.19). Exemplary, the pushouts (31) and (22) are composed in Isabelle/HOL.

```
interpret "31+22": pushout_diagram "interf r_2" c21.H "lhs r_2" H_1
                                   "c21.c ∘→ j_2" b_2 s_1 "h_1 ∘→ i_2"
  using pushout_composition[OF
    "31.flip_diagram" "22.flip_diagram" ] by assumption
```

The final pushout (5) is constructed and the pushouts are rearranged and vertically composed as depicted in Fig. 17. Isabelle is able to discharge the goal at this point automatically as we instantiated all required locales. The sequential independence follows from the construction. This finishes the second stage of the proof. □

In developing the proofs, we manually laid out the proof structure by carefully instantiating the locales with the correct parameters in the right order. This process was challenging due to the large number of type parameters involved. However, Isabelle's typechecker provided some support by catching errors when we attempted to instantiate the parameters in the wrong order, helping to guide the proof development process.

Despite this assistance, the overall proof structure was developed manually, with tactics being employed only for specific subgoals. The manual approach allowed us to maintain control over the proof flow and ensured that the reasoning remained clear and understandable. While the type parameters added complexity to the proof, they also provided a valuable safeguard against potential inconsistencies that could arise from incorrectly ordered instantiations which could lead to long debugging.

5. Related Work

Isabelle/HOL was used by Strecker for interactive reasoning about graph transformations [Str18]. A major difference to our work is that he introduces a form of graph transformation that does not fit with any of the established approaches, such as the double-pushout approach. As a consequence, his framework cannot draw on existing theory. Another difference is that [Str18] focuses on verifying first-order properties of some form of graph programs while the current paper is concerned with formalising and proving fundamental results of the DPO theory. Strecker's formalisation fixes node and edge identifiers as natural numbers, while we keep them abstract. Similar to our development, Isabelle's locale mechanism is employed.

Our formalisation of graphs follows the work of Noschinski [Nos15], where records are used to group components and locales to enforce properties such as the well-formedness of graphs or morphisms. The main objective of [Nos15] is to formalise and prove fundamental results of classical graph theory, such as Kuratowski's theorem.

Stark [Sta16] developed an "object-free" definition of categories in Isabelle/HOL, which simplifies the specification of a category and allows functors and natural transformations to be defined as functions satisfying certain axioms. Here, the primary focus is on efficiently representing abstract algebra in higher-order logic. A limitation of this approach is that a special zero element had to be defined. As a result, the formalization prevents the construction of a discrete category over the entire universe, and instead requires a large enough base type. We followed a more traditional approach in formalizing the category theory-related concepts but had to work around the single universe and explicit quantification limitations discussed earlier.

da Costa Cavalheiro et al. [dCCFR17] use the Event-B specification method and its associated theorem prover to reason about double-pushout and single-pushout graph transformations, where rules can have attributes and negative application conditions. Event-B is based on first-order logic and typed set theory. Different from our approach, [dCCFR17] gives only a non-formalised proof for the equivalence between the abstract definition of pushouts and the set-theoretic construction. In contrast, we formalise both the abstract and the operational view and prove their correspondence using Isabelle/HOL. As Event-B is based on first-order logic, the properties that can be expressed and verified are quite limited. For it is known that non-local properties of finite graphs cannot be specified in first-order logic [Lib04]. This restriction does not apply to our formalisation, as we can make full use of higher-order logic.

6. Conclusion

In this paper, we have significantly advanced our formalisation of basic DPO graph transformation theory using the Isabelle proof assistant, relying on its higher-order logic instantiation. We have thoroughly elaborated on two pivotal results of the DPO theory: the uniqueness of direct derivations and the Church-Rosser theorem. Our discussion includes a series of critical lemmata that build towards the final proof.

We also delve into the technical aspects of our formalisation, including the application of total and partial functions, finite sets, and maps (`fset` and `fmap`). Our approach strikes a balance between expressiveness and automation support. While the finite sets (maps) offer advantages in certain proofs, their lack of a comprehensive lemma set needs significant additional implementation and proof work on our part. Moreover, the idea of using a unified identifier for both nodes and edges, though initially appealing, led to increased complexity in our construction, as detailed in Section 2. This complexity extended to the entire process of gluing and deletion, affecting the properties we proved using this construction.

Looking forward, our aim is to broaden our methodology to include attributed DPO graph transformation, as outlined in [HP16]. Our long-term goal is to develop a GP 2 proof assistant, enabling the interactive verification of individual graph programs. This tool may leverage the proof calculus presented in [WP21], offering a significant advancement in the field of graph program verification.

Acknowledgment

We are grateful to Brian Courthoute, Annegret Habel, and Thomas Türk for discussions on the topics of this paper.

References

[ADGR07] Jeremy Avigad, Kevin Donnelly, David Gray, and Paul Raff. A formally verified proof of the prime number theorem. *ACM Transactions on Computational Logic*, 9(1):2, 2007. `doi:10.1145/1297658.1297660`.

[Bal21] Clemens Ballarin. *Tutorial to locales and locale interpretation*, 2021. URL: `https://isabelle.in.tum.de/doc/locales.pdf`, `arXiv:https://isabelle.in.tum.de/doc/locales.pdf`.

[BH18] Achim D. Brucker and Michael Herzberg. A Formal Semantics of the Core DOM in Isabelle/HOL. In *Companion Proceedings of the The Web Conference 2018*, WWW '18, page 741–749. International World Wide Web Conferences Steering Committee, 2018. `doi:10.1145/3184558.3185980`.

[CCP22] Graham Campbell, Brian Courtehoute, and Detlef Plump. Fast rule-based graph programs. *Science of Computer Programming*, 214, 2022. `doi:10.1016/j.scico.2021.102727`.

[dCCFR17] Simone André da Costa Cavalheiro, Luciana Foss, and Leila Ribeiro. Theorem proving graph grammars with attributes and negative application conditions. *Theoretical Computer Science*, 686:25–77, 2017. `doi:10.1016/j.tcs.2017.04.010`.

[Dí20] Javier Díaz. Finite map extras. *Archive of Formal Proofs*, October 2020. `https://isa-afp.org/entries/Finite-Map-Extras.html`.

[EEPT06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. Springer, 2006. `doi:10.1007/3-540-31188-2`.

[Ehr79] Hartmut Ehrig. Introduction to the algebraic theory of graph grammars. In *Proc. Graph-Grammars and Their Application to Computer Science and Biology*, volume 73 of *Lecture Notes in Computer Science*, pages 1–69. Springer-Verlag, 1979. `doi:10.1007/BFb0025714`.

[EK76] Hartmut Ehrig and Hans-Jörg Kreowski. Parallelism of manipulations in multidimensional information structures. In *Proc. Mathematical Foundations of Computer Science (MFCS 1976)*, volume 45 of *Lecture Notes in Computer Science*, pages 284–293. Springer, 1976.

[EK79] Hartmut Ehrig and Hans-Jörg Kreowski. Pushout-properties: An analysis of gluing constructions for graphs. *Mathematische Nachrichten*, 91:135–149, 1979.

[Gon07] Georges Gonthier. The four colour theorem: Engineering of a formal proof. In *Proc. Asian Symposium on Computer Mathematics (ASCN 2007)*, volume 5081 of *Lecture Notes in Computer Science*, page 333. Springer, 2007. `doi:10.1007/978-3-540-87827-8\_28`.

[HAB$^+$15] Thomas Hales, Mark Adams, Gertrud Bauer, Dat Tat Dang, John Harrison, Truong Le Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Thang Tat Nguyen, Truong Quang Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason Rute, Alexey Solovyev, An Hoai Thi Ta, Trung Nam Tran, Diep Thi Trieu, Josef Urban, Ky Khac Vu, and Roland Zumkeller. A formal proof of the Kepler conjecture. *Forum of Mathematics, Pi*, 5, 2015. `doi:10.1017/fmp.2017.1`.

[HK13] Brian Huffman and Ondřej Kunčar. Lifting and transfer: A modular design for quotients in Isabelle/HOL. In *Certified Programs and Proofs: Third International Conference, CPP 2013, Proceedings 3*, pages 131–146, 2013. `doi:10.1007/978-3-319-03545-1_9`.

[HMP01] Annegret Habel, Jürgen Müller, and Detlef Plump. Double-pushout graph transformation revisited. *Mathematical Structures in Computer Science*, 11(5):637–688, 2001. `doi:10.17/S0960129501003425`.

[HP16] Ivaylo Hristakiev and Detlef Plump. Attributed graph transformation via rule schemata: Church-Rosser theorem. In *Software Technologies: Applications and Foundations – STAF 2016 Collocated Workshops, Revised Selected Papers*, volume 9946 of *Lecture Notes in Computer Science*, pages 145–160. Springer, 2016. `doi:10.1007/978-3-319-50230-4_11`.

[HR04] Michael Huth and Mark Dermot Ryan. *Logic in Computer Science - Modelling and Reasoning about Systems*. Cambridge University Press, 2nd edition, 2004.

[HUW14] John Harrison, Josef Urban, and Freek Wiedijk. History of Interactive Theorem Proving. In *Handbook of the History of Logic*, volume 9, pages 135–214. Elsevier, 2014. `doi:10.1016/B978-0-444-51624-4.50004-6`.

[KEH$^+$09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David A. Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In *Proc. Symposium on Operating Systems Principles (SOSP 2009)*, pages 207–220. ACM, 2009. `doi:10.1145/1629575.1629596`.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009. `doi:10.1145/1538788.1538814`.

[Lib04] Leonid Libkin. *Elements of Finite Model Theory*. Texts in Theoretical Computer Science. Springer, 2004. `doi:10.1007/978-3-662-07003-1`.

[LS04] Stephen Lack and Paweł Sobociński. Adhesive categories. In *Proc. Foundations of Software Science and Computation Structures (FoSSaCS 2004)*, volume 2987 of *Lecture Notes in Computer Science*, pages 273–288. Springer, 2004. `doi:10.1007/978-3-540-24727-2\_20`.

[NK14] Tobias Nipkow and Gerwin Klein. *Concrete semantics: with Isabelle/HOL*. Springer, 2014. `doi:10.1007/978-3-319-10542-0`.

[Nos15] Lars Noschinski. A graph library for Isabelle. *Mathematics in Computer Science*, 9(1):23–39, 2015. `doi:10.1007/s11786-014-0183-z`.

[Plu16] Detlef Plump. Reasoning about graph programs. In *Proc. Computing with Terms and Graphs (TERMGRAPH 2016)*, volume 225 of *Electronic Proceedings in Theoretical Computer Science*, pages 35–44, 2016. `doi:10.4204/EPTCS.225.6`.

[PNW19] Lawrence C. Paulson, Tobias Nipkow, and Makarius Wenzel. From LCF to Isabelle/HOL. *Formal Aspects of Computing*, 31(6):675–698, 2019. `doi:10.1007/s00165-019-00492-1`.

[Ros75] Barry K. Rosen. Deriving graphs from graphs by applying a production. *Acta Informatica*, 4:337–357, 1975.

[SP22] Robert Söldner and Detlef Plump. Towards mechanised proofs in double-pushout graph transformation. In *Proc. International Workshop on Graph Computation Models (GCM 2022)*, volume 374 of *Electronic Proceedings in Theoretical Computer Science*, pages 59–75, 2022. `doi:10.4204/EPTCS.374.6`.

[SP23] Robert Söldner and Detlef Plump. Mechanised DPO Theory: Uniqueness of Derivations and Church-Rosser Theorem. In *Proc. International Conference on Graph Transformation (ICGT 2023)*, Lecture Notes in Computer Science, pages 123–142. Springer, 2023. doi:10.1007/978-3-031-36709-0_7.

[Sta16] Eugene W. Stark. Category theory with adjunctions and limits. *Archive of Formal Proofs*, June 2016. https://isa-afp.org/entries/Category3.html, Formal proof development.

[Str18] Martin Strecker. Interactive and automated proofs for graph transformations. *Mathematical Structures in Computer Science*, 28(8):1333–1362, 2018. doi:10.1017/S096012951800021X.

[SW09] Norbert Schirmer and Makarius Wenzel. State spaces - the locale way. In *Proc. International Workshop on Systems Software Verification (SSV 2009)*, volume 254 of *Electronic Notes in Theoretical Computer Science*, pages 161–179, 2009. doi:10.1016/j.entcs.2009.09.065.

[Wen] Makarius Wenzel. *The Isabelle/Isar Reference Manual*. https://isabelle.in.tum.de/doc/isar-ref.pdf.

[Wen99] Markus Wenzel. Isar — A generic interpretative approach to readable formal proof documents. In *Theorem Proving in Higher Order Logics (TPHOLs 1999)*, volume 1690 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 1999. doi:10.1007/3-540-48256-3_12.

[WP21] Gia S. Wulandari and Detlef Plump. Verifying graph programs with monadic second-order logic. In *Proc. International Conference on Graph Transformation (ICGT 2021)*, volume 12741 of *Lecture Notes in Computer Science*, pages 240–261. Springer, 2021. doi:10.1007/978-3-030-78946-6_13.