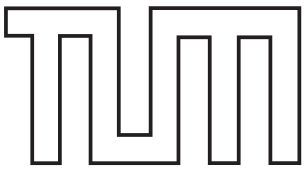


Fakultät für Informatik
Lehrstuhl für Logik und Verifikation

Formalizing Graph Theory and Planarity Certificates

Lars Noschinski

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzende:	Univ.-Prof. Dr. Susanne Albers
Prüfer der Dissertation:	
1.	Univ.-Prof. Tobias Nipkow, Ph.D.
2.	Hon.-Prof. Dr. Kurt Mehlhorn Universität des Saarlandes

Die Dissertation wurde am 03.11.2015 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 05.03.2016 angenommen.

Abstract

This thesis studies the treatment of graphs in a proof assistant and analyzes certificates for planarity of graphs.

For a proof assistant to be of practical use, a comprehensive library of formalized mathematics is crucial. Graph theory is often a weak point: isolated results have been formalized, but a common vocabulary is missing.

This thesis describes a theory of directed graphs in the Isabelle/HOL proof assistant and therefore provides such a common vocabulary. This theory is used for reasoning about directed and undirected graphs. In particular, two planarity certificates are formalized.

The second contribution of this thesis is the verification of programs checking these planarity certificates. Tools dedicated to program verification are weak when it comes to proving the underlying theory of a program. On the other hand, proof assistants lack a convenient language for the actual verification. This gap is closed by a new technique for structuring proof obligations arising from program verification.

Zusammenfassung

Diese Dissertation befasst sich mit der Darstellung von Graphen in Theorembeweisern und der formalen Analyse von Zertifikaten für Planarität.

Für die praktische Einsetzbarkeit eines Theorembeweisers ist eine umfassende Bibliothek formalisierter Mathematik entscheidend. Graphentheorie ist hier häufig ein Schwachpunkt: Vereinzelte Resultate wurden formalisiert, aber es fehlt eine gemeinsame Basis.

Diese Arbeit beschreibt eine Formalisierung von gerichteten Graphen im Theorembeweiser Isabelle/HOL und stellt damit eine solche gemeinsame Basis zur Verfügung. Diese Formalisierung wird für Beweise über gerichtete und ungerichtete Graphen genutzt. Insbesondere werden zwei Planaritätszertifikate formalisiert.

Der zweite Beitrag dieser Arbeit ist die Verifikation von Programmen, die ein solches Planaritätszertifikat überprüfen. Eine Schwäche von dedizierten Werkzeugen zur Programmverifikation ist die Formalisierung der einem Programm zugrunde liegenden Mathematik. Auf der anderen Seite fehlt Theorembeweisern eine geeignete Sprache für die eigentliche Verifikation. Diese Lücke wird durch eine neue Technik geschlossen, die es ermöglicht, die bei der Programmverifikation entstehenden Beweisverpflichtungen zu strukturieren.

Acknowledgments

I am deeply obliged to the people who accompanied me during my work on this thesis..

Tobias Nipkow. I thoroughly enjoyed researching and teaching under your supervision.

The other members of the theorem proving group, now known as the chair for logic and verification, past and present. Jasmin Blanchette, Sascha Böhme, Julian Brunner, Lukas Bulwahn, Manuel Eberl, Holger Gast, Florian Haftmann, Johannes Hölzl, Brian Huffman, Lars Hupel, Fabian Immler, Alexander Krauss, Ondřej Kunčar, Peter Lammich, Andrei Popescu, Dmitriy Traytel, Thomas Türk, and Makarius Wenzel: It was a pleasure working and discussing with you. I learned a lot.

Eleni Nikolaou-Weiß and Silke Müller. You were the backbone of our group.

Johannes Hölzl, Lars Hupel, Fabian Immler, and Peter Lammich. For reading the drafts of my thesis and many helpful comments.

Peter Lammich. For many engaging discussions.

My parents. For being there from the beginning.

My wife. For Everything.

Thank you, everyone!

Contents

1. Introduction

Graph theory is an area of mathematics with a wide range of applications, in particular in computer science. Its beginning as a field of mathematical studies dates back to Leonhard Euler. The Königsberg bridge problem proved by Euler is perhaps one of the most well-known problems in graph theory. A graph is commonly depicted as a set of vertices or nodes, connected by edges. Drawing a graph on a piece of paper immediately poses the question whether this is possible without edges crossing other edges, leading to the notion of planarity.

This thesis formalizes a theory of directed graphs and two definitions of planarity. It then moves into the domain of algorithms and provides verified implementations of programs checking (non-)planarity certificates. A new technique for structuring proofs supports the program verification.

1.1. Outline

This thesis is structured as follows.

Chapter 2 presents a proof of the Girth-Chromatic Number theorem. It uses the probabilistic method pioneered by Erdős [25] which is notable for deriving definite results by using tools from probability theory. The results build on a very minimal formalization of undirected graphs.

Chapter 3 introduces a library for reasoning about directed graphs. Building on a very general representation of digraphs, it serves as the foundation for the rest of the thesis. It also contains a characterization of directed Euler graphs and incorporates the result of Chapter 2.

Chapter 4 formalizes two characterizations of planarity. I give a result for the relation between these two characterizations and proof the correctness of a decision procedure for planarity.

Chapter 5 recapitulates the concept certifying algorithms. I formalize a certificate for the non-planarity of graphs and verify two programs checking these certificates.

Chapter 6 considers proof obligations arising in program verification. Such obligations are unstructured and large, making their proofs hard to read and maintain. This chapter introduces a method for structuring such proofs.

Chapter 7 completes the work of Chapter 5 with the verification of a program checking planarity certificates. The verification serves as a case study for the method presented in Chapter 6.

Chapter 8 recapitulates the results and makes suggestions for future work.

The chapters are written to be read in chronological order. Exceptions are Chapter 2 and Chapter 6, which are independent of the other chapters. Readers not interested in the technicalities of program verification can skip Chapter 6, while readers only interested in my method for structured program verification proofs may confine themselves to Chapter 6 and Chapter 7.

All results have been formalized or implemented in the Isabelle/HOL proof assistant.

1.2. Publications

This thesis builds upon the following four publications, ordered chronologically:

[65] Lars Noschinski. "Proof Pearl: A Probabilistic Proof for the Girth-Chromatic Number Theorem". In: *Interactive Theorem Proving*. LNCS. 2012

[67] Lars Noschinski, Christine Rizkallah, and Kurt Mehlhorn. "Verification of Certifying Computations through AutoCorres and Simpl". In: *NASA Formal Methods*. LNCS. Apr. 2014, pp. 46–61

[60] Lars Noschinski. "A Graph Library for Isabelle". In: *Mathematics in Computer Science* 9.1 (June 2014), pp. 23–39

[66] Lars Noschinski. "Towards Structured Proofs for Program Verification (Ongoing Work)". Isabelle Workshop (Interactive Theorem Proving). 2014

Chapter 2 is based on [65] and Chapter 3 extends [60]. I reuse parts of [67] with the permission of Christine Rizkallah and Kurt Mehlhorn for Chapter 5. An early prototype of the technique described in Chapter 6 has been presented at the Isabelle Workshop at ITP 2014 [66]. The results in Chapter 4 and Chapter 7 have not been published before.

All results have been proven (or, for Chapter 6, implemented) in the Isabelle/HOL theorem prover. Most of these formalizations have been published in the *Archive of Formal Proofs* (*AFP*).

[61] Lars Noschinski. "A Probabilistic Proof of the Girth-Chromatic Number Theorem". In: *Archive of Formal Proofs* (Feb. 2012). Formal proof development

[63] Lars Noschinski. "Graph Theory". In: *Archive of Formal Proofs* (Apr. 2013). Formal proof development

[62] Lars Noschinski. "Generating Cases from Labeled Subgoals". In: *Archive of Formal Proofs* (July 2015). Formal proof development

[64] Lars Noschinski. "Planarity Certificates". In: *Archive of Formal Proofs* (Oct. 2015). Formal proof development

The verification of the C implementation in Chapter 5 has not yet been published in the AFP, as it has external dependencies not yet included in the AFP.[1]

[1] For now, the verification of the *C* implementation is available at http://www21.in.tum.de/~noschinl/Non_Planarity_Certificate/.

1.3. Isabelle/HOL

All chapters of this thesis are concerned with formalizing (or implementing) a result in Isabelle/HOL. Isabelle/HOL (or just Isabelle) is a proof assistant based on polymorphic higher-order logic [58], following the LCF tradition. That is, every theorem accepted by Isabelle follows from the axioms of the logic and a few primitive inference rules. If one trusts the kernel implementing these inferences, one can be confident that the derived theorems are correct.

I mostly use standard mathematical notation with a few changes as described below. Variables are written in an *italic* font and constant and function symbols in sans-serif. HOL types include type variables ($\alpha, \beta, \ldots$), function types ($\alpha \to \beta$), tuples ($\alpha \times \beta$), sets (α set), lists (α list), and number types ($\mathbb{N}$, $\mathbb{R}$). The type α of a term t is indicated as $t : \alpha$ and function application is written $f\ t$. To apply a function $f : \alpha \to \beta$ pointwise to a set $X : \alpha$ set, I use the notation $f\ X \coloneqq \{f\ x \mid x \in X\}$.

List variables usually are named *xs*, *ys* and so on. Lists are constructed from Nil ([]) and Cons ($x :: xs$) and list literals are denoted by $[x_1, \ldots, _n]$. Concatenation of lists is written $xs +\!\!+ ys$, and hd and tl decompose a list such that $\mathsf{hd}\ (x :: xs) = x$ and $\mathsf{tl}\ (x :: xs) = xs$. The expression map $f\ xs$ applies the function f to every element of a list *xs* and set *xs* refers to the set of elements of *xs*. As on sets, $x \in xs$ is the membership test and $xs \setminus ys$ is the list derived from *xs* by removing all elements of *ys*. For a single element list, I also write $xs \setminus x$ instead of $xs \setminus [x]$.

New types can be defined as subsets of other types (**typedef**), as records (i.e., tuples with named fields, **record**), or inductively (**datatype**). I write record literals as tuples and the selector functions have the same name as the field. The logical connectives follow the standard notation from mathematics. Lemmas are denoted $[\![P_1; \ldots; P_n]\!] \Longrightarrow Q$ where $P_1, \ldots, P_n$ are premises and Q is the conclusion. The $[\![\ldots]\!]$ notation is shorthand for $P_1 \Longrightarrow \ldots \Longrightarrow P_n \Longrightarrow Q$. Terms bound by λ-abstraction or a quantifier extend as far to the right as possible.

The two reasoning styles commonly used in Isabelle are rewriting and natural deduction. Rewriting is mostly performed with the *simplifier*, which uses conditional equations and discharges the conditions again by simplification. Natural deduction rules are applied by hand or by one of various classical reasoning tools. Isabelle's most popular proof methods combine both styles. The proof tools can be configured to solve common goals automatically by declaring appropriate rules as simplification rules (for rewriting) or introduction or elimination rules (for natural deduction).

Locales Isabelle provides *locales* [6] for modularization of locales. They allow to structure formal proof developments by providing a local collection of constants and assumptions, as in the example below:

> **locale** L = **fixes** $c : \tau$ **assumes** $P\ c$
> **locale** K = **fixes** $d : \tau$ **assumes** $Q\ d$

The first command declares a new locale L with a local constant c of type τ and introduces a local assumption $P\ c$. All type variables occurring in τ are also fixed locally. The number of constants and assumptions is arbitrary. The fixed constants are called the *parameters* of the locale. Once defined, a locale can be extended with theorems and definitions, relying implicitly

on the locale parameters and assumptions. We can also declare a locale inheriting from another locale:

locale L' = L + **fixes** $c' : \tau'$ **assumes** $P'\ c\ c'$

The parameters of L' are c and c', and the assumptions $P\ c$ and $P'\ c\ c'$. The locale L' also inherits all the theorems and definitions from L.

It is possible to access the theorems and definitions in a locale from the outside. In this case, the locale assumptions become explicit assumptions of the theorems and locale parameters (and type variables) are universally quantified. For constants defined in the locale, the locale parameters are now passed explicitly. To improve clarity, such parameters will be denoted as a subscript (if not clear from the context).

To apply a locale to concrete parameters, it can be interpreted. An *interpretation* of L instantiates the parameter c with a concrete value t. Moreover, one can give arbitrary equations $s = s'$ to refine the interpretation. After proving that $P\ t$ and the additional equations hold, all the lemmas of L become available as in the locale, with c instantiated to t and s rewritten to s'. An *embedding* is an interpretation of a locale in another locale. The command

sublocale $\mathsf{K} \subseteq \mathsf{L}\ t$ **where** $s = s'$

embeds K into L, instantiating d with t and replacing s by s'. The terms t, s, s' may contain the parameters of K.

2. A Probabilistic Proof of the Girth-Chromatic Number Theorem

The Girth-Chromatic number theorem is a theorem from graph theory, stating that graphs with arbitrarily large girth and chromatic number exist. In this chapter, I formalize a probabilistic proof of this theorem in the Isabelle/HOL theorem prover, closely following a standard textbook proof, and use this to explore the use of the probabilistic method in a theorem prover. An earlier version of this article was published in the proceedings of the ITP 2012 [65].

A common method to prove the existence of some object is to construct it explicitly. The probabilistic method, which I explain below, is an alternative if an explicit construction is hard. In this paper, I explore whether the use of the probabilistic method is feasible in a modern interactive theorem prover. Consider the Girth-Chromatic Number theorem from graph theory. The girth g is the size of the shortest cycle in the graph. The chromatic number χ of a graph is the minimal number of colors which is needed to color the vertices in a way such that adjacent vertices have different colors. The Girth-Chromatic number theorem then states that there are graphs with a large girth and a high chromatic number, i.e., for an arbitrary natural number ℓ there exists a graph G with both $\chi\, G > \ell$ and $g\, G > \ell$. On a first glance, these properties seem contradictory: For a fixed number of vertices, the complete graph containing all edges has the largest chromatic number. On the other hand, if the cycles are large, such a graph is locally acyclic and hence locally 2-colorable. This discrepancy makes it hard to inductively define a graph satisfying this theorem.

Indeed, the first proof of this theorem given by Erdős [25] used an entirely non-constructive approach: Erdős constructed a probability space containing all graphs of order n. Using tools from probability theory he then proved that, for large enough n, randomly choosing a graph yields a witness for the Girth-Chromatic Number theorem with a non-zero probability. Hence, such a graph exists. It took 9 more years before a constructive proof was given by Lovász [48].

This use of probability theory is known as *probabilistic method*. Erdős and Rényi are often considered the first conscious users of this method and developed it in their theory of Random Graphs [13, 26]. Other applications include Combinatorics and Number Theory. In this work, I explore how well this technique works in a modern theorem prover.

The well-known Girth-Chromatic Number theorem is one of the early applications of Random Graphs and often given as an example of applications of the probabilistic method. The proof presented here follows the one given by Diestel [18]. The Isabelle/HOL theory files of this formalization can be found in the Archive of Formal Proofs [61].

This chapter is structured as follows: Section 2.1 defines basic graph properties and operations and Section 2.2 introduces a probability space on graphs. In Section 2.3, I describe how asymptotic properties are handled. Section 2.4 gives a high-level description of the proof of the Girth-Chromatic Number theorem before the formal proof is described in Section 2.5. I reflect

on this formalization and review related work in Section 2.6.

2.1. Modeling Graphs

I consider undirected and loop-free *graphs* $G = (V, E)$. V and E are sets of vertices and edges, respectively. Edges are represented as sets of vertices. A *loop* is an edge from a vertex to itself, that is a singleton set $\{u\}$. I assume that $V \subseteq \mathbb{N}$. Although I do not explicitly require graphs to be finite, only finite graphs are relevant in this formalization.

I use V_G and E_G to refer to the vertices and edges of a graph G. The *order* of a graph is the cardinality of its vertex set. A graph is called *wellformed*, if every edge connects exactly two distinct vertices of the graph. This is expressed by the following predicate:

$$\text{wellformed } G \coloneqq (\forall e \in \mathsf{E}_G.\ |e| = 2 \wedge e \subseteq \mathsf{V}_G)$$

A *walk* is a sequence of vertices such that consecutive vertices are connected by an edge. I represent walks as non-empty lists of vertices and define the edge list of a walk recursively. The length of a walk (denoted by $||\cdot||$) is the length (denoted by $|\cdot|$) of its edge list. A *cycle* is a closed walk with at least three edges where each vertex occurs at most once (cycles with length 2 are uninteresting, because the same edge is used twice). It is represented as a walk where first and last vertex are equal, but each other vertex occurs at most once. Note that a cycle of length k can be represented by $2k$ different walks.

$$\begin{aligned}
\text{walk-edges } [] &\coloneqq [] \\
\text{walk-edges } [x] &\coloneqq [] \\
\text{walk-edges } (x :: y :: xs) &\coloneqq \{x, y\} :: \text{walk-edges } (y :: xs)
\end{aligned}$$

$$\begin{aligned}
||p|| &\coloneqq |\text{walk-edges } p| \\
\text{walks } G &\coloneqq \{p \mid p \neq [] \wedge \text{set } p \subseteq \mathsf{V}_G \wedge \text{set } (\text{walk-edges } p) \subseteq \mathsf{E}_G\} \\
\text{cycles } G &\coloneqq \{p \in \text{walks } G \mid 3 \leq ||p|| \wedge \text{distinct } (\text{tl } p) \wedge \text{hd } p = \text{last } p\}
\end{aligned}$$

The *girth* g of a graph is the length of its shortest cycle; the girth of a graph without cycles will be denoted by ∞. The set of natural numbers extended with ∞, denoted $\mathbb{N}_\infty$, forms a complete lattice, so the girth of a graph can be defined as the infimum of the length of its cycles:

$$g\ G \coloneqq \inf_{p \in \text{cycles } G} ||p||$$

A *vertex coloring* is a partition of the vertices of a graph, such that adjacent vertices are in different subsets. The *chromatic number* χ is the size of the smallest such partition. The power set of X is written 2^X.

$$\begin{aligned}
\text{colorings } G \coloneqq\ & \{C \subseteq 2^{\mathsf{V}_G} \mid \bigcup C = \mathsf{V}_G \\
& \wedge (\forall V_1, V_2 \in C.\ V_1 \neq V_2 \implies V_1 \cap V_2 = \varnothing) \\
& \wedge (\forall V \in C.\ V \neq \varnothing \wedge (\forall u, v \in V.\ \{u, v\} \notin \mathsf{E}_G))\} \\
\chi\ G \coloneqq\ & \inf_{C \in \text{colorings } G} |C|
\end{aligned}$$

These definitions suffice to state the Girth-Chromatic Number theorem. For an arbitrary $\ell \in \mathbb{N}$ holds:

$$\exists G.\ \text{wellformed}\ G \wedge \ell < \chi\ G \wedge \ell < g\ G$$

However, a few auxiliary definitions are needed; most notably the notion of an independent set and the independence number α. Two vertices $u, v \in \mathsf{V}_G$ are *independent* if $u \neq v$ and $\{u, v\} \notin \mathsf{E}_G$. A set $V \subseteq \mathsf{V}_G$ is an *independent set* if and only if all the vertices in V are independent. The *independence number* α is the size of the largest independent set.

$$\begin{aligned} \mathsf{E}_V &:= \{\{u, v\} \mid u, v \in V \wedge u \neq v\} \\ \text{independent-sets}\ G &:= \{V \subseteq \mathsf{V}_G \mid \mathsf{E}_V \cap \mathsf{E}_G = \varnothing\} \\ \alpha\ G &:= \sup_{V \in \text{independent-sets}\ G} |V| \end{aligned}$$

Here, E_V is the set of all (non-loop) edges on V. I also write E_n for $\mathsf{E}_{\{1,\ldots,n\}}$. Note that each coloring is a disjoint union of independent sets. This gives us a lower bound for the chromatic number:

Lemma 2.1 (Lower Bound for $\chi\ G$). *For all graphs G, $|G|/\alpha\ G \leq \chi\ G$.*

Proof. A vertex coloring is a partition of G and a disjoint union of non-empty independent sets. The size of these sets is bounded by $\alpha\ G$ and hence a vertex coloring consists of at least $|G|/\alpha\ G$ sets. □

Removing Short Cycles Besides the usual graph theoretic definitions, we will need an operation to remove all short cycles from a graph. For a number k, a *short cycle* is a cycle with length at most k:

$$\text{short-cycles}\ (G, k) := \{c \in \text{cycles}\ G \mid ||c|| \leq k\}$$

We remove the short cycles by repeatedly removing a vertex from a short cycle until no short cycle is left. To remove a vertex from a graph, all edges adjacent to this vertex are also removed.

$$\begin{aligned} G - \{u\} &:= (V_G \setminus \{u\}, E_G \setminus \{e \in E_G \mid u \in e\}) \\ \text{choose-v}\ (G, k) &:= \varepsilon\ (\lambda u.\ \exists p \in \text{short-cycles}\ (G, k).\ u \in p) \\ \text{kill-short}\ (G, k) &:= \begin{cases} \varnothing & \text{if short-cycles}\ (G, k) = \varnothing \\ \text{kill-short}\ (G - \{\text{choose-v}\ (G, k)\}, k) & \text{else} \end{cases} \end{aligned} \tag{2.1}$$

To select an arbitrary vertex, I use Hilbert's choice operator ε. Given a predicate P, the expression $\varepsilon\ P$ denotes some element satisfying P, if such an element exists, or an arbitrary element from the domain of P otherwise.

Equation (2.1) describes a recursive function which does not terminate on some infinite graphs. However, an (underspecified) function with these equation can easily be defined by the **partial_function** command of Isabelle. To prove some properties about the graphs computed by kill-short, a specialized induction rule is useful.

Lemma 2.2 (Induction rule for kill-short). *Let k be a natural number. If for all graphs H both*

$$\text{short-cycles}\ (H,k) = \varnothing \implies P\ (H,k)$$

and

$$\begin{aligned}\text{finite}\ (\text{short-cycles}\ (H,k)) \wedge \text{short-cycles}\ (H,k) \neq \varnothing \\ \wedge P\ (H - \{\text{choose-v}\ (H,k)\}) \implies P\ (H,k)\end{aligned}$$

hold, then $P\ (G,k)$ holds for all finite *graphs G.*

The canonical induction principle for kill-short has finite H as premise for the second rule. Replacing this premise with finite (short-cycles (G,k)) strengthens the induction hypothesis and makes the induction principle more convenient, for example to prove Lemma 2.5 below. With this induction rule, one can easily prove the following theorems about kill-short for finite graphs G:

Lemma 2.3 (Large Girth). *The girth of* kill-short (G,k) *exceeds k, i.e.,*

$$k < g\ (\text{kill-short}\ (G,k)).$$

Lemma 2.4 (Order of Graph). kill-short (G,k) *removes at most as many vertices as there are short cycles, i.e.,*

$$|\mathsf{V}_G| - |\mathsf{V}_{\text{kill-short}\ (G,k)}| \leq |\text{short-cycles}\ (G,k)|\ .$$

Lemma 2.5 (Independence Number). *Removing the short cycles does not increase the independence number, i.e., α* (kill-short $(G,k)) \leq \alpha\ G$.

Proof. Removing a vertex v does not increase the independence number: two vertices of $G - \{v\}$ are adjacent if and only if they are adjacent in G. □

Lemma 2.6 (Wellformedness). *Removing short cycles preserves wellformedness, i.e.,*

$$\text{wellformed}\ G \implies \text{wellformed}\ (\text{kill-short}\ (G,k)).$$

2.2. Probability Space

There are a number of different probability models which are commonly used for the analysis of random graphs. To prove the Girth-Chromatic number theorem, I consider a series of probability spaces $\mathcal{G}_n$ of graphs of order n, for n going to infinity. $\mathcal{G}_n$ consists of all graphs G with $\mathsf{V}_G = \{1,\ldots,n\}$ and $E_G \subseteq \mathsf{E}_n$. A randomly chosen graph $G \in \mathcal{G}_n$ contains an edge $e \in \mathsf{E}_n$ with probability p_n. As V_G is fixed to $\{1,\ldots,n\}$, a graph $G \in \mathcal{G}_n$ is uniquely defined by its edges; so instead of a space of graphs $\mathcal{G}_n$, I define a space $\mathcal{E}_n$ of edge sets. This turns out to be slightly more convenient.

To define such a probability space in a canonical way, for each edge in E_n one defines a probability space on $\{0,1\}$, such that 1 occurs with probability p_n and 0 occurs with probability $1 - p_n$. Then, $\mathcal{G}_n$ is identified with the product of these probability spaces.

This construction is supported by Isabelle's extensive library on probability theory [37]. However, the elements of the product space of probability spaces are functions $2^{2^{\mathbb{N}}} \to \{0,1\}$ which are only specified on E_n. Identifying these with edge sets triggers some amount of friction in a theorem prover. To avoid this, I construct a probability space on edge sets without using the product construction. This is easily possible as $\mathcal{E}_n$ is finite for all n.

For the definition of $\mathcal{E}_n$, consider the following: in the setting above, the probability that a randomly chosen edge set contains a fixed edge e is p_n, the probability of the negation is $1 - p_n$. The probabilities of the edges are independent, so the probability that a randomly chosen edge set is equal to a fixed set $E \subseteq \mathsf{E}_n$ is $p_n^{|E|} \cdot (1-p_n)^{|\mathsf{E}_n - E|}$, i.e., the product of the edge probabilities.

Definition 2.7 (Probability Space on Edges). *Let $n \in \mathbb{N}$ and $q \in \mathbb{R}$ with $0 \leq q \leq 1$. Let $f\ E = q^{|E|} \cdot (1-q)^{|\mathsf{E}_n - E|}$ for all $E \in 2^{\mathsf{E}_n}$. Lift f to sets of edges by $\mathcal{P}_{n,q}\ S = \sum_{E \in S} f\ E$ for all $S \subseteq 2^{\mathsf{E}_n}$. Then $\mathcal{E}_{n,q} = (2^{\mathsf{E}_n}, \mathcal{P}_{n,q})$ is the probability space with domain E_n and probability function $\mathcal{P}_{n,q}$. When a function $p : \mathbb{N} \to \mathbb{R}$ is given from the context, we also write $\mathcal{E}_n$ and $\mathcal{P}_n$ for $\mathcal{E}_{n,p\,n}$ and $\mathcal{P}_{n,p\,n}$.*

Isabelle's probability library provides a locale for probability spaces. One option to specify such a space is by giving a finite domain X and a probability function μ with the following properties: For each $x \in X$ holds $0 \leq \mu\ x$ and $\sum_{x \in D} \mu\ x = 1$. When we can show that those two properties hold for $X = 2^{\mathsf{E}_n}$ and $\mu = \mathcal{P}_n$ in Definition 2.7, then Isabelle's locale mechanism transfers all lemmas about probability spaces to $\mathcal{E}_n$. In particular, the following lemma is needed:

Lemma 2.8 (Markov's Inequality). *Let $P = (X, \mu)$ be a probability space, $c \in \mathbb{R}$ and $f : X \to \mathbb{R}$ such that $0 < c$ and for all $x \in X$ holds $0 \leq f\ x$. Then*

$$\mu\ \{x \in X \mid c \leq f\ x\} \leq 1/c \cdot \sum_{x \in X} f\ x \cdot \mu\ \{x\}.$$

Obviously $0 \leq \mathcal{P}_n$. Hence, $\mathcal{E}_n$ is a probability space by the following lemma.

Lemma 2.9 (Sum of Probabilities Equals 1). *Let S be a finite set. Then for all $0 \leq p \leq 1$ holds*

$$\left(\sum_{A \subseteq S} p^{|A|} \cdot (1-p)^{|S-A|} \right) = 1.$$

A similar lemma describes the probability of certain sets of edge sets.

Lemma 2.10 (Probability of Cylinder Sets). *Let n and q such that $\mathcal{E}_{n,q}$ is a probability space and $\mathsf{cyl}_n\ (A, B) \coloneqq \{E \subseteq \mathsf{E}_n \mid A \subseteq E \wedge B \cap E = \varnothing\}$ the set of all edge sets containing A but not B. Then $\mathcal{P}_{n,q}\ (\mathsf{cyl}_n\ (A, B)) = q^{|A|} \cdot (1-q)^{|B|}$ for all disjoint $A, B \subseteq \mathsf{E}_n$.*

2.3. Handling Asymptotics

As mentioned in Section 2.2, we consider a series of probability spaces. In many cases, it suffices if a property P holds after a finite number of steps, i.e., $\exists k.\ \forall n > k.\ P\ n$. Often, one can avoid dealing with these quantifiers directly. For example, to prove

$$(\exists k_1.\ \forall n > k_1.\ P\ n) \wedge (\exists k_2.\ \forall n > k_2.\ Q\ n) \implies \exists k_3.\ \forall n > k_3.\ R\ n$$

it suffices to show $\exists k.\ \forall n > k.\ P\ n \wedge Q\ n \implies R\ n$ or even $\forall n.\ P\ n \wedge Q\ n \implies R\ n$ instead. However, such a rule would be inconvenient to use in practice, as proof automation tends to destroy the special form of the quantifiers. This can be prevented by using a specialized constant instead of the quantifiers. In Isabelle, such a constant (with suitable lemmas) is already available in the form of *filters* [14] and the eventually predicate. Filters generalize the concept of a sequence and are used in topology and analysis to define a general notion of convergence; they can also be used to express quantifiers [11]. In rough terms, a filter is a non-empty set of predicates closed under conjunction and implication and eventually is the membership test. I use eventually with the filter

$$\text{sequentially} := \{P \mid \exists k.\ \forall n > k.\ P\ n\}$$

as kind of a universal quantifier. This fits nicely Isabelle's definition of a limit:

$$\lim_{n\to\infty} f\ n = c \iff \forall \varepsilon > 0.\ \text{eventually}\ (\lambda n.\ |f\ n - c| < \varepsilon)\ \text{sequentially}$$

The formula $\exists k.\ \forall n > k.\ P\ n$ is equivalent to eventually P sequentially. I will denote this as $\forall^\infty n.\ P\ n$ or write "$P\ n$ holds for large n". The following three rules enable us to reason about eventually as described above:

$$\frac{\forall n.\ k < n \implies P\ n}{\forall^\infty n.\ P\ n} \qquad (\textit{eventually-sequentiallyI})$$

$$\frac{\forall^\infty n.\ P\ n \quad \forall^\infty n.\ P\ n \implies Q\ n}{\forall^\infty n.\ Q\ n} \qquad (\textit{eventually-rev-mp})$$

$$\frac{\forall^\infty n.\ P\ n \quad \forall^\infty n.\ Q\ n \quad \forall n.\ (P\ n \wedge Q\ n) \implies R\ n}{\forall^\infty n.\ R\ n} \qquad (\textit{eventually-elim2})$$

Apart from rule *eventually-sequentiallyI*, these hold for the eventually predicate in general. The rule *eventually-elim2* is actually just a convenience rule, which can be easily derived from the other two rules by dropping the condition $k < n$.

2.4. Proof Outline

I now give a high-level outline of the proof. Let ℓ be a natural number. Recall the statement of the Girth-Chromatic Number theorem:

$$\exists G.\ \text{wellformed}\ G \wedge \ell < \chi\ G \wedge \ell < g\ G$$

Instead of working with the chromatic number, we will use the independence number α. By Lemma 2.1, it suffices to show

$$\exists G.\ \text{wellformed}\ G \wedge \alpha\ G < |G|/\ell \wedge \ell < g\ G$$

to prove the Girth-Chromatic Number theorem. Estimating probabilities for α is easier than for χ as an independent set is a cylinder set, cf. Lemma 2.10.

The basic idea of the probabilistic proof is to show that, for large enough n, a randomly graph $G \in \mathcal{G}_n$ (respectively set of edges $E \in \mathcal{E}_n$) has the desired properties with a non-zero probability. A reasonable approach would be to search for a probability function p_n, for which one can show $\mathcal{P}_n\{G \mid g\, G \leq \ell\} + \mathcal{P}_n\{G \mid \alpha\, G \geq n/\ell\} < 1$. This would imply that a graph G satisfying neither $g\, G \leq \ell$ nor $\chi\, G \leq \ell$ exists. Such a graph G would satisfy the Girth-Chromatic number property. It turns out that a probability function with this property does not exist [18].

However, this idea can be salvaged: Instead of searching for a graph which satisfies the Girth-Chromatic Number property, we search for a graph which almost has this property, i.e., we allow a small number of short cycles. By choosing p_n correctly, we can show the following property

$$\mathcal{P}_n\{G \mid n/2 \leq |\text{short-cycles}\ (G, \ell)|\} + \mathcal{P}_n\{G \mid 1/2 \cdot n/\ell \leq \alpha\, G\} < 1$$

and obtain a graph with at most $n/2$ short cycles and an independence number less than $1/2 \cdot n/\ell$ (i.e., $2\ell < \chi\, G$ by Lemma 2.1). From this graph, we remove a vertex from every short cycle. The resulting graph then has large girth and the chromatic number is still large.

2.5. The Proof

As a first step, we derive an upper bound for the probability that a graph has at least $1/2 \cdot n/k$ independent vertices. The syntax $G_{n,E}$ is a shortcut for the graph $(\{1, \ldots, n\}, E)$

Lemma 2.11 (Probability for many Independent Edges). *Given $n, k \in \mathbb{N}$ such that $2 \leq k \leq n$, we have*

$$\mathcal{P}_n\{E \subseteq \mathsf{E}_n \mid k \leq \alpha\, G_{n,E}\} \leq \binom{n}{k}(1 - p_n)^{\binom{k}{2}}\ .$$

Proof. Holds by a simple combinatorial argument and Lemma 2.10. □

Lemma 2.12 (Almost never many Independent Edges). *Assume that $0 < k$ and $\forall^\infty n.\ 0 < p_n \wedge p_n < 1$. If in addition $\forall^\infty n.\ 6k \cdot \ln n/n \leq p_n$ holds, then there are almost never more then $1/2 \cdot n/k$ independent vertices in a graph, i.e.,*

$$\lim_{n\to\infty} \mathcal{P}_n\{E \subseteq \mathsf{E}_n \mid 1/2 \cdot n/k \leq \alpha\, G_{n,E}\} = 0$$

Proof. With Lemma 2.11. □

Then we compute the expected number of representatives of cycles of length k in a graph. Together with Markov's Lemma, this will provide an upper bound of

$$\mathcal{P}_n\{E \in \mathcal{E}_n \mid n/2 \leq |\text{short-cycles}\ (G_{n,E}, \ell)|\}.$$

Lemma 2.13 (Mean Number of k-Cycles). *If $3 \leq k < n$, then the expected number of paths of length k describing a cycle is*

$$\left(\sum_{E \in \mathcal{E}_n} \left|\{c \in \text{cycles}\ G_{n,E} \mid k = ||c||\}\right| \cdot \mathcal{P}_n\{E\}\right) = \frac{n!}{(n-k)!} \cdot p^k$$

We arrive at our final theorem:

Theorem 2.14 (Girth-Chromatic Number)**.** *Let ℓ be a natural number. Then there is a (well-formed) graph G, such that $\ell < g\ G$ and $\ell < \chi\ G$:*

$$\exists G.\ \text{wellformed}\ G \wedge \ell < g\ G \wedge \ell < \chi\ G$$

To prove this, we fix $p_n = n^{\varepsilon - 1}$ where $\varepsilon = 1/(2\ell)$ and assume without loss of generality that $3 \leq \ell$. These assumptions hold for all of the following propositions. With Lemma 2.13, we can derive an upper bound for the probability that a random graph of size n has more than $n/2$ short cycles:

Proposition 2.15.

$$\forall^\infty n.\ \mathcal{P}_n\ \{E \subseteq \mathsf{E}_n \mid n/2 \leq |\text{short-cycles}\ (G_{n,E}, \ell)|\} \leq 2(\ell - 2)n^{\varepsilon\ell - 1}$$

As this converges to 0 for n to infinity, eventually the probability will be less than $1/2$:

Proposition 2.16.

$$\forall^\infty n.\ \mathcal{P}_n\ \{E \subseteq \mathsf{E}_n \mid n/2 \leq |\text{short-cycles}\ (G_{n,E}, \ell)|\} < 1/2$$

Similarly, with these choices, the conditions of Lemma 2.12 are satisfied:

Proposition 2.17.

$$\forall^\infty n.\ \mathcal{P}_n\ \{E \subseteq \mathsf{E}_n \mid 1/2 \cdot n/\ell \leq \alpha\ G_{n,E}\} < 1/2$$

Therefore, the sum of these probabilities will eventually be smaller than 1 and hence, with a non-zero probability, there is a graph with only few short cycles and a small independence number:

Proposition 2.18. *There exists a graph $G \in \mathcal{G}_n$ with a small independence number and a small number of short cycles, i.e., $1/2 \cdot n/\ell > \alpha\ G$ and $n/2 > |\text{short-cycles}\ (G, \ell)|$.*

Removing the short cycles turns this graph into a witness for the Girth-Chromatic Number theorem. This completes the proof of Theorem 2.14.

Proposition 2.19. *Let G be a graph obtained from Proposition 2.18. Then the graph $H :=$ kill-short (G, ℓ) satisfies $\ell < g\ H$ and $\ell < \chi\ H$. Moreover, H is wellformed.*

Proof. By Lemmas 2.3–2.6 and 2.1. □

Actually, we almost proved an even stronger property: The probabilities in Proposition 2.16 and Proposition 2.17 converge both to 0, so almost all graphs satisfy the condition of Proposition 2.18. Hence, almost every graph can be turned into a witness for the Girth-Chromatic Number theorem by removing the short cycles. This is typical for many proofs involving the probabilistic method.

2.6. Discussion

In this chapter, I formally proved the Girth-Chromatic Number theorem from graph theory, closely following the text book proof. The whole proof consists of just 84 theorems (1439 lines of Isabelle theories), split into three files and is therefore quite concise. Around 41 of these lemmas are of general interest, reasoning about reals with infinity and some combinatorial results. Partly, these have been added to the Isabelle distribution. Moreover, 18 lemmas are given about basic graph theory and the core proof of the theorem consists of the remaining 25 lemmas (around 740 lines). For the core proof, I mostly kept the structure of the text book proof, so auxiliary propositions only needed for one lemma are not counted separately.

The result looks straightforward, but there are some design choices I like to discuss. In an early version of this formalization, edges were represented by an explicit type of two-element sets. However, it turned out that this made some proof steps a lot more complicated: Isabelle does not support subtyping, so defining a two-element-set type yields a new type disjoint from sets with a partial type constructor. When one needs to refer to the vertices connected by an edge, this partiality makes reasoning harder. This easily offsets the little gain an explicit edge type gives in our setting (wellformedness is only explicitly required in two theorems). Indeed, Erdős and Hajnal [23] have shown that the theorem can be generalized to hypergraphs or set systems.

One should note that our definition of the chromatic number is not as obviously correct as it appears from the first glance: for an infinite graph G, $\chi\ G = 0$. This is due to the standard definition of cardinality in Isabelle mapping infinite sets to 0. I decided not to care about this, as this formalization is only about finite graphs (and the final theorem assures a positive chromatic number anyway).

The main reason I decided to use $\mathbb{N}_\infty$ instead of $\mathbb{N}$ was to be able to give a natural definition of the girth – without infinity, an extra predicate to handle the "no cycles" case would be necessary. A nice side effect is that α and χ are easier to handle, as infimum and supremum are also defined on empty and infinite sets. However, as a result of this choice, real numbers including infinity ($\mathbb{R} \cup \{\infty, -\infty\}$) are needed. If these had not been already available as a library, it would probably have been easier to avoid infinity altogether and special-case the girth of acyclic graphs.

The use of eventually turned out to be quite rewarding. For the proofs for Lemma 2.12 and the propositions for Theorem 2.14, I quite often collect a number of facts holding for large n and eliminate them like in Section 2.3. This allowed for more elegant proofs, as it removed the need of keeping track of irrelevant lower bounds.

Now, which capabilities are needed to use the probabilistic method in a theorem prover? Obviously some amount of probability theory. Different fragments of probability theory are now formalized in many theorem provers, including HOL4, HOL-light, PVS, Mizar and Isabelle [22, 37, 41, 47, 54]. Surprisingly, for the proof presented here, not much more than Markov's Inequality is required. For other proofs, more stochastic vocabulary (like variance and independence) is needed. One example of such a proof was formalized by Hupel [40], which builds upon the probability spaces defined in this formalization.

If one makes the step from finite to infinite graphs (for example to prove the Erdős-Rényi theorem that almost all countably infinite graphs are isomorphic [24, 68]), infinite products of probability spaces are required. To our knowledge, the only formalization of these is found in

Isabelle [37].

Furthermore, good support for real arithmetic including powers, logarithms and limits is needed. Isabelle has this, but proving inequalities on complex terms remains tedious as often only very small proof steps are possible. However, the calculational proof style [10] (inspired by Mizar) is very helpful here.

In the future, an automated reasoner for inequalities over real-value functions like MetiTarski [1] might be useful. However, the set of a few example inequalities from our proof which L. Paulson kindly tested for me was outside the reach of MetiTarski.

Related Work Proofs with the probabilistic method often lead to randomized algorithms. Probably the first formalization in this area is Hurd's formalization of the Miller-Rabin primality test [42]; other work on this topic is available in Coq [4]. A constructive proof of a theorem similar to the Girth-Chromatic Number theorem was formalized by Rudnicki and Stewart in Mizar [70].

There are a few general formalizations of undirected graphs available in various theorem provers, for example [15, 16, 45]; but often proof developments rather use specialized formalizations of certain aspects of graph theory [30, 57] to ease the proof. For the Girth-Chromatic Number theorem, the common definition of graphs as pairs of vertices and edges seems quite optimal. The Girth-Chromatic Number theorem does not rely on any deep properties about graphs and the formalization of graphs we give here is rather straightforward. In Chapter 3 I present a general graph library for directed graphs in Isabelle/HOL. Directed graphs can be used to model undirected graphs, so I will discuss in Section 3.5 how the Girth-Chromatic Number theorem can be transferred to the setting of this library.

Conclusion I gave a concise (and, to my knowledge, the first) formal proof for the well-known Girth-Chromatic Number theorem and explored the use of the probabilistic method in a theorem prover, which worked well for this theorem. It will be interesting to see whether this continues to hold true for more involved theorems. An interesting example for this could be Lovász Local Lemma: Many probabilistic proofs show not only that the probability is non-zero, but even that it tends to 1 for large graphs. The Local Lemma can be used to show that a property holds with a positive, but very small probability. This enables some combinatorical results, for which no proof not involving this lemma is known [3].

3. A Graph Library for Isabelle

In contrast to other areas of mathematics such as calculus, number theory, or probability theory, there is currently no standard library for graph theory in the Isabelle/HOL proof assistant. I present a formalization of directed graphs and essential related concepts. The library supports general infinite directed graphs (digraphs) with labeled and parallel arcs, but care has been taken not to complicate reasoning on more restricted classes of digraphs.

I use this library to formalize a characterization of Euler Digraphs. An earlier version of this chapter has been published in Mathematics in Computer Science [60].

3.1. Introduction

Modern proof assistants usually include a library covering many areas of mathematics. The formalizations in these libraries serve as a common basis for proof developments, enabling the user to reuse results proved by others, and also keeping the developments aligned. For graph theory, despite its many applications, the situation is a different: no standard formalization has evolved yet.

In the HOL family of proof assistants, there have been a number of formalizations of specific graph-theoretic algorithms and results, but none has evolved into a general library. The formalization of Dijkstra' shortest path algorithm in Isabelle/HOL [59] covers only the essentials needed for this algorithm. Wong's work on railway networks in HOL [83] covers walks, trails, paths, degrees, union and operations to insert or delete vertices and arcs. However, I am not aware of any work building upon it. Chou [16] formalizes undirected graphs in HOL, including concepts like walks, paths, trails, reachability, connectedness, bridges, and rooted trees, as well as operations to combine graphs while preserving tree-ness. A similar set of notions is formalized in Coq by Duprat [21], using an inductive graph definition. The NASA PVS libraries cover both undirected and directed graphs, but do not consider parallel arcs [15]. Mizar contains a comparatively large amount of graph theory, which is however split on six different formalizations of graphs. This covers formalizations of the algorithms by Dijkstra, Prim and Ford-Fulkerson, as well as a characterization of undirected Euler graphs[56]. Planar graphs have been formalized in the Flyspeck project[57] and in the proof of the Four-Color-Theorem[30], using specialized graph representations.

In this article, I present a formalization of directed graphs in Isabelle/HOL. This is the first attempt at a general graph library for this proof assistant. In part, my formalization resembles the one chosen by Wong [83] for his work on railway networks. My formalization works in a more abstract setting, which enables re-use in different applications. In particular, it allows the user to choose a graph implementation depending on the needs of the application.

The contribution of my work is a comprehensive set of basic graph theory vocabulary. The library is general enough to reason about all common classes of directed graphs, including those

with parallel and labeled arcs. This work has been motivated by the verification effort on the LEDA graph library [52]. The LEDA library contains a number certifying graph algorithms and my library has been successfully used to verify the certificates generated by some of these algorithms (see [2, 67, 69], Chapter 5, and Chapter 7).

A major goal of this formalization is to enable convenient reasoning about digraphs, including good proof automation. For this reason, I emphasize the choice of an appropriate representation of digraphs and walks. For efficient proving it is important that the proof heuristics can solve frequently occurring proof obligations automatically. Therefore, I stress where the aim of better automation leads to a particular formulation. As a particular challenge, graph theory distinguishes between simple digraphs (without parallel arcs) and multi-digraphs (with parallel arcs). Textbooks often prefer the former, as they are easier to handle. Many applications, on the other hand, use or allow the use of parallel arcs. My goal is that the option to work with multi-digraphs should not increase the proof effort needed on digraphs.

Based on a representation of directed multi-graphs, I have formalized the concepts of finite, loop-free and simple digraphs, walks, paths, cycles, (symmetric) reachability, isomorphisms, degree of vertices, (induced) subgraphs, (strong) connectedness and connected components, and trees and spanning trees. I also provide operations for the union of graphs, adding and removing arcs and vertices, and many lemmas to combine these concepts. As a case study, I have formalized a characterization of directed Euler trails. In later chapters, I also use this library to present some results on planarity. Moreover, this library has been used to formalize cycle checking algorithms in [27] and in the verification of checkers for certifying algorithms in the LEDA graph library. The library is available in the Archive of Formal Proofs [63]. The formalization described here consists of around 6500 lines of Isabelle theories.

The chapter is organized as follows: Section 3.2 describes the representation of digraphs and discusses possible alternatives. Section 3.3 contains a selection of formalized concepts and Section 3.4 presents a case study about Euler digraphs. Section 3.5 investigates the interoperability with other formalizations of graph theory. The chapter concludes with a discussion of the results in Section 3.6.

3.2. Representation

In this section, I present some representations of digraphs and discuss alternatives. For graph theoretic terms and definitions, I mostly follow Bang-Jensen and Gutin [7]. I will note where the definitions deviate. A *digraph* consists of vertices V and arcs A. An *arc* connects two vertices, going from the *tail* (or source) to the *head* (or target). Common representations of arcs are $A \subseteq V \times V$ for simple digraphs or $A \subseteq V \times L \times V$ for multi-digraphs, where L is some set of labels.

I represent a digraph as a 4-tuple (V, A, t, h), where A is a set of abstract values and $t, h : A \to V$ map an arc to its tail resp. head. This formulation is also found in some textbooks [77].

Definition 3.1 (Type of directed graphs). *A directed graph is a 4-tuple:*

record (β, α) dg = verts : β set, arcs : α set, tail : $\alpha \to \beta$, head : $\alpha \to \beta$

I abbreviate verts G, arcs G, tail G u *with* V_G, A_G, $u_{\mathsf{t},G}$, *and* $u_{\mathsf{h},G}$, *respectively. Two graphs are* compatible *if the projection functions are extensionally equal.*[1]

$$\text{compatible } G\ H \coloneqq \text{tail } G = \text{tail } H \wedge \text{head } G = \text{head } H$$

I always use α and β for the types of arcs and vertices, respectively.

By choosing appropriate values for α, tail, and head, we can select the representation appropriate to a problem domain. This includes the representations with $A \subseteq V \times V$ or $A \subseteq V \times L \times V$ above. Another example are digraphs with $\beta = \alpha = \mathbb{N}$, which sometimes occur in the verification of imperative programs (where the number is an index into some data structure or the heap). The latter representation has been used in the verification of C programs dealing with graphs [67].

I use locales to structure the different classes of digraphs. The most basic class are *well-formed* digraphs. A digraph is well-formed if the endpoints of all arcs are vertices of the graph. Unless noted otherwise, the term "digraph" is used to refer to well-formed directed graphs. This is in contrast to Bang-Jensen and Gutin [7], who use "digraph" to what we will call simple digraphs. Hence, I state lemmas in the wf-digraph locale. Functions and predicates are defined in the pre-digraph locale. As this locale does not have any assumptions, the definitions of these functions can be used even if well-formedness has not yet been proven.

Definition 3.2 (Well-formed Graph).

locale pre-digraph = **fixes** $G : (\beta, \alpha)$ dg
locale wf-digraph = pre-digraph +
assumes $\forall a \in \mathsf{A}_G.\ a_{\mathsf{t},G} \in \mathsf{V}_G$ **and** $\forall a \in \mathsf{A}_G.\ a_{\mathsf{h},G} \in \mathsf{V}_G$

In contrast to Bang-Jensen and Gutin [7], I do not exclude the null graph (i.e., an empty set of vertices is allowed). This is slightly more convenient in the case studies and it makes digraphs closed under deletion of vertices or intersection. Harary and Read [36] discuss the merits of allowing this graph.

Often digraphs are required to be finite (i.e., the sets of arcs and vertices are finite), loop-free (i.e., head and tail of each arc are distinct) or free of parallel arcs (i.e., there is at most one arc for each pair of vertices). I also sometimes model undirected graphs as symmetric graphs (i.e., the relation described by the arcs is symmetric) or as bidirected graphs.

Definition 3.3 (Bidirected Graph). (G, π) *is a* bidirected digraph *if* G *is a wellformed digraph and* π *pairs each arc with an unique reverse arc.*

locale bidirected-digraph = wf-digraph G **for** G +
fixes $\pi : \beta \to \beta$
assumes $\forall a \in \mathsf{A}_G.\ a \in \mathsf{A}_G \longleftrightarrow \pi\ a \neq a$
assumes $\forall a.a \in \mathsf{A}_G.\ \pi\ (\pi\ a) = a$
assumes $\forall a.a \in \mathsf{A}_G.\ (\pi\ a)_{\mathsf{t},G} = a_{\mathsf{h},G}$ *(this implies* $(\pi\ a)_{\mathsf{h},G} = a_{\mathsf{t},G}$*)*

G *is* bidirectable, *if there is a permutation* π *such that* (G, π) *is bidirected.*

[1] If we restricted ourselves to a fixed vertex type β, we could use Isabelle's type class mechanism to define the projection functions for each type α once and for all, instead of defining them on a per-graph basis.

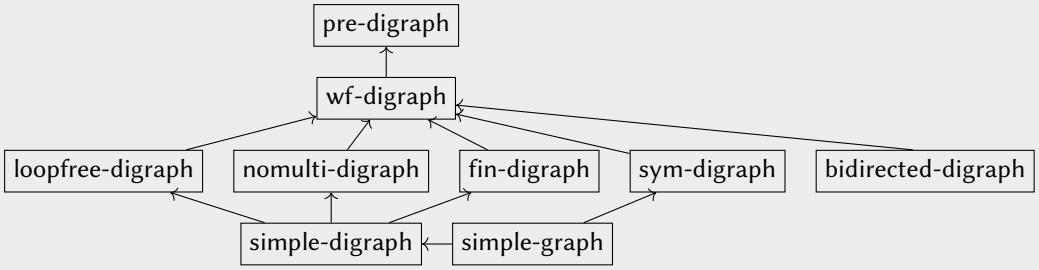


Figure 3.1.: Hierarchy of digraph locales. The arrows describe an inheritance relationship.

For each of the graph properties described above, I introduce a separate locale (Figure 3.1). These locales can be combined. For example *simple digraphs* are defined as the union of finite, loop-free digraphs without parallel arcs. Locales for other combinations can be easily defined. As a general rule, lemmas are proven in the most general locale possible, so that the more specialized locales inherit these.

These graph locales make no assumption about the relation between β and α. As a result, it is not always possible to add an arc to connect two vertices of a graph: As an pathological example, consider α = unit, the type with a single element (), and the graph $(\{a, b\}, \{()\})$ where $a \neq b$ and the arc () has tail a and head b. This graph cannot be extended with an arc from b to a as there is no second value in unit. To avoid such cases, one needs to place additional restrictions on α, tail and head.

3.2.1. Specialized Representations

In this section, I show how one can regain the simple representations discarded in the previous section. In particular, I explain how locales can be used to eliminate the projection functions of (β, α)dg. For many representations, these are fixed for all graphs, so it should not be necessary to give these explicitly for every graph. I demonstrate this approach with the usual representation of graphs without parallel arcs.

Definition 3.4 (Pair Digraph).

record β pair-dg = pverts : β set, parcs : $(\beta \times \beta)$ set
locale pair-wf-digraph =
fixes G : β pair-dg
assumes $\forall e \in$ parcs G. fst $e \in$ pverts G **and** $\forall e \in$ parcs G. snd $e \in$ pverts G

A pair digraph G can be embedded into the type $(\beta, \beta \times \beta)$ dg) *as follows:*

$$\overline{G} := (\text{pverts } G, \text{parcs } G, \text{fst}, \text{snd})$$

The newly defined locale can be embedded into wf-digraph:

sublocale pair-wf-digraph $\subseteq$ wf-digraph $\overline{G}$ **where**
tail $\overline{G}$ = fst **and** head $\overline{G}$ = snd **and** $\mathsf{V}_{\overline{G}}$ = pverts G **and** $\mathsf{A}_{\overline{G}}$ = parcs G

The additional equations are used by the locale mechanism to rewrite the theorems embedded from wf-digraph and remove every occurrence of the selector functions of the dg in these theorems. This is important for automation: without it, the embedded lemmas are often cumbersome to use. For example, consider an introduction rule $[\![a \in \mathsf{A}_G]\!] \Longrightarrow a_{\mathrm{t},G} \in \mathsf{V}_G$, stating that the tail of an arc is a vertex of the digraph. It would have to be rewritten manually before it can be applied to a goal of the form $[\![\ldots]\!] \Longrightarrow \mathsf{fst}\ a \in \mathsf{V}_G$, which is the natural form for the pair digraphs.

Due to this rewriting, the embedded definitions of some functions do not depend on G anymore. An example is the predicate cas, which tests whether a sequence of arcs is consistent, i.e., their endpoints fit together:

Definition 3.5 (Consistent Arc Sequence)**.**

$$\begin{aligned} \mathsf{cas}_G\ u\ []\ v &\coloneqq (u = v) \\ \mathsf{cas}_G\ u\ (a :: as)\ v &\coloneqq (a_{\mathrm{t},G} = u \wedge \mathsf{cas}_G\ (a_{\mathrm{h},G})\ as\ v) \end{aligned}$$

In pair-wf-digraph, the second equation is rewritten to:

$$\mathsf{cas}_{\overline{G}}\ u\ (a :: as)\ v = (\mathsf{fst}\ a = u \wedge \mathsf{cas}_{\overline{G}}\ (\mathsf{snd}\ a)\ as\ v)$$

So, if G and H are of type β pair-dg, then $\mathsf{cas}_{\overline{G}}$ and $\mathsf{cas}_{\overline{H}}$ are the same function, denoted by two different terms. I replace such functions by a new constant without the digraph parameter. This makes proofs involving these functions easier for automated proof tools.

When using pair graphs, I will implicitly use the embedding function wherever necessary. In Isabelle, this is done using coercive subtyping [75].

If one wanted to introduce a more general notion of graphs later on, one could use a similar approach to identify concepts on digraphs with the more general concepts. A candidate for such a step would be mixed graphs [7] as a common basis for directed and undirected graphs or the vertex/edge/link-representation chosen by Chou [16], which can also express hypergraphs. This would allow us to share common concepts between these different types of graphs.

3.3. Operations and Properties

In this section, I present a number of basic concepts I formalized.

3.3.1. Walks and Related Concepts

In textbooks, a *walk* is a finite alternating sequence $u_1 a_1 u_2 a_2 \ldots u_{k-1} a_{k-1} u_k$ of vertices u_i and arcs a_i of the digraph such that a_i has tail u_i and head u_{i+1}. I omit the vertices and represent a walk as list of arcs. The expression $\mathsf{awalk}_G\ u\ p\ v$ denotes that p is a walk form u to v. The predicate cas (Definition 3.5) separates the consistency from the membership properties. This is useful for proofs involving compatible graphs, as consistency of a walk is preserved over such graphs.

Definition 3.6 (Walk, Vertices of a Walk)**.**

$$\mathsf{awalk}_G\ u\ p\ v \coloneqq (u \in \mathsf{V}_G \wedge \mathsf{set}\ p \subseteq \mathsf{A}_G \wedge \mathsf{cas}_G\ u\ p\ v)$$

This is equivalent to the following recursive definition used for the simplifier:

$$\mathsf{awalk}_G\ u\ []\ v \coloneqq u = v \wedge u \in \mathsf{V}_G$$
$$\mathsf{awalk}_G\ u\ (a :: as)\ v \coloneqq a \in \mathsf{arcs}_G \wedge u = a_{\mathsf{t},G} \wedge \mathsf{awalk}_G\ a_{\mathsf{h},G}\ as\ v$$

The function $\mathsf{aw\text{-}verts}_G$ *computes the vertices of a walk. To deal with the empty list, an explicit start vertex must be given.*

$$\mathsf{aw\text{-}verts}_G\ u\ [] \coloneqq [u]$$
$$\mathsf{aw\text{-}verts}_G\ u\ (a :: as) \coloneqq a_{\mathsf{t},G}\ ::\ \mathsf{aw\text{-}verts}_G\ a_{\mathsf{h},G}\ as$$

I abbreviate $\mathsf{last}\ (\mathsf{aw\text{-}verts}_G\ u\ p)$ *by* $\mathsf{awl}_G\ u\ p$*.*

All walks consisting of a single vertex are represented by the empty list. This caused no problems in our experiments, as the awalk predicate mentions the start and end vertices of a walk.

Arc lists have a nice concatenation property: the concatenation of two lists is a walk if and only if both lists are already walks on their own and there exists a common endpoint. For the rewrite rule, this common endpoint is denoted by awl, instead of an using existential quantifier, as the simplifier's ability to deal with quantifiers is limited.

Lemma 3.7. *For the concatenation of two walks, the following equality holds:*

$$\mathsf{awalk}_G\ u\ (p +\!\!+ q)\ v \iff \mathsf{awalk}_G\ u\ p\ (\mathsf{awl}_G\ u\ p) \wedge \mathsf{awalk}_G\ (\mathsf{awl}_G\ u\ p)\ q\ v$$

If there is an awalk_G *predicate for* P*, the following conditional equation can remove the* awl *terms:*

$$[\![\mathsf{awalk}_G\ u\ p\ v]\!] \Longrightarrow \mathsf{awl}_G\ u\ p = v. \tag{3.1}$$

Automatic rewriting with (3.1) will loop, if v is already of the form $\mathsf{awl}_G\ u\ p$. As this rule is very useful otherwise, I implement a general simplification procedure (*simproc*) which prevents rewriting in this case. A simproc is a plugin for Isabelle's simplifier, which can compute rewrite rules for a given pattern on demand. For that, I define a new constant $\mathsf{NOMATCH} : \tau \to \tau \to \mathsf{bool}$ with $\mathsf{NOMATCH}\ t\ p \coloneqq \mathsf{True}$ and configure the simplifier to not rewrite the arguments of this constant. If the simproc encounters a NOMATCH term, it tests whether the term t matches the pattern p. This is done on a purely syntactic level, so the procedure, in contrast to normal rewrite rules, is able to distinguish between terms which are logically equal. If v does not match p, the simproc returns the equation $\mathsf{NOMATCH}\ t\ p = \mathsf{True}$, otherwise it fails, preventing the simplifier from rewriting the term.

This allows to state (3.1) in a form which is safe for automatic rewriting:

$$[\![\mathsf{awalk}_G\ u\ p\ v;\ \mathsf{NOMATCH}\ (\mathsf{awl}_G\ u\ p)\ v]\!] \Longrightarrow \mathsf{awl}_G\ u\ p = v.$$

The order of the assumptions is important: When applying this rule, the simplifier first tries to prove $\mathsf{awalk}_G\ u\ p\ v$. This causes v to be instantiated. If now $\mathsf{awl}_G\ u\ p$ matches v, the simplifier fails to prove the assumptions of this rule and the rule is not applied. The NOMATCH simproc is now used in various theories both in the Isabelle distribution and the AFP. The use of a purely syntactic constant to guide the simplifier has been motivated by ACL2's rewriter [39].

There are two reasons to choose arc lists instead of alternating sequences: for alternating sequences, concatenation either removes the last vertex of the first sequence or the first vertex of the second sequence. For example, $(u, a, v, b, y) = (u, a, v) \frown (x, b, y)$ being a walk does not imply that (u, a, v) and (x, b, y) are walks, so Lemma 3.7 does not hold without additional knowledge about the components. A similar problem occurs with all representations that include (at least) one explicit start or end vertex. The other reason is that finite arc sequences map to Isabelle lists in a natural way, which allows us to use Isabelle's well-developed theory of lists, including the various induction schemes.

Like alternating sequences, the representation of walks as a list of vertices lacks the nice concatenation property of arc lists. For this reason, I usually found it more convenient to reason with arc lists than with vertex lists. Nevertheless, the formalization also includes some basic facts about walks as vertex lists and their relation to arc lists.

In his work, Wong [83] also represents walks as list of arcs, but excludes the single-vertex walk, deviating from most textbooks. This sometimes simplifies the formalization, since the vertices of a walk can be denoted without an explicit start vertex, but has the disadvantage that lemmas about the decomposition of walks become more complex.

Apart from walks, I also formalized trails, paths, and cycles as walks with additional properties:

Definition 3.8 (Trails, Paths, and Cycles)**.**

$$\begin{aligned}\mathsf{atrail}_G\ u\ p\ v &\coloneqq \mathsf{awalk}_G\ u\ p\ v \wedge \mathsf{distinct}\ p\\ \mathsf{apath}_G\ u\ p\ v &\coloneqq \mathsf{awalk}_G\ u\ p\ v \wedge \mathsf{distinct}\ (\mathsf{aw\text{-}verts}_G\ u\ p)\\ \mathsf{cycle}_G\ p &\coloneqq \exists u.\ \mathsf{awalk}_G\ u\ p\ u \wedge p \neq [] \wedge \mathsf{distinct}\ (\mathsf{tl}\ (\mathsf{aw\text{-}verts}_G\ u\ p))\end{aligned}$$

The function tl *removes the first element of a list and* distinct *states that list does not contain duplicates.*

I follow Diestel [18] and Volkmann [77] here and allow cycles consisting of a single arc, i.e., loops. Bang-Jensen and Gutin Bang-Jensen and Gutin [7] require at least two arcs.

It is often necessary to relate arcs and vertices of a walk. The following lemma serves as the basis for lemmas to split a walk based on a vertex property, for example to shorten a walk to a path by removing parts where the vertices are not distinct.

Lemma 3.9. *If* $\mathsf{awalk}_G\ u\ p\ v$ *and* $\mathsf{aw\text{-}verts}_G\ u\ p = xs +\!\!+ [y] +\!\!+ ys$ *hold, then*

$$\begin{aligned}&\exists q, r.\ \mathsf{awalk}_G\ u\ q\ y \wedge \mathsf{awalk}_G\ y\ r\ v \wedge p = q +\!\!+ r \wedge\\ &\mathsf{aw\text{-}verts}_G\ u\ q = xs +\!\!+ [y] \wedge \mathsf{aw\text{-}verts}_G\ y\ r = [y] +\!\!+ ys.\end{aligned}$$

For a walk going from one set of vertices to another, one can extract the arc which marks the transition with the following lemma:

Lemma 3.10. *Let P, Q be predicates on vertices. Then holds:*

$$[\![\mathsf{awalk}_G\ u\ p\ v;\ p \neq [];\ \forall w \in (\mathsf{aw\text{-}verts}_G\ u\ p).\ P\ w \vee Q\ w;\ P\ u;\ Q\ v]\!] \\ \Longrightarrow\ \exists a \in p.\ P\ a_{\mathrm{t},G} \wedge Q\ a_{\mathrm{h},G}$$

If one only wants to know whether a vertex v is reachable from another vertex u, an inductively defined relation is more convenient than the proposition $\exists p.\ \mathsf{awalk}_G\ u\ p\ v$. The basic relation here is the adjacency relation $\mathsf{adj}_G : (\beta \times \beta)\ \mathsf{set}$, also written as $u \rightarrow_G v$ (or just $u \rightarrow v$, if G is clear from the context). One can use the usual operations on relations to define various reachability properties. For reflexive reachability relations, the domain must be restricted to V_G. I use an inductively defined predicate rtrancl-on equivalent to $\mathsf{adj}_G^* \cap \mathsf{V}_G \times \mathsf{V}_G$ (on well-formed digraphs). This is also written as $u \rightarrow^* v$. The library contains the usual transitivity rules as well as induction rules progressing left-to-right and right-to-left:

$$[\![u \rightarrow^* v;\ u \in \mathsf{V}_G \Longrightarrow P\ u;\ (\forall x, y.\ u \rightarrow^* x \wedge x \rightarrow y \wedge P\ x) \Longrightarrow P\ y]\!] \ \Longrightarrow\ P\ v$$
$$[\![u \rightarrow^* v;\ v \in \mathsf{V}_G \Longrightarrow P\ v;\ (\forall x, y.\ x \rightarrow y \wedge y \rightarrow^* v \wedge P\ y) \Longrightarrow P\ x]\!] \ \Longrightarrow\ P\ u$$

3.3.2. Operations and Properties of Graphs

Digraphs can be modified by adding and removing arcs and vertices. This always yields a well-formed digraph. I do not provide a generic function to build an arc from two vertices for the abstract graph type. This operation depends on α and often needs additional assumptions on the projection functions.

Definition 3.11 (Adding and Removing Arcs and Vertices)**.**

$$\begin{aligned} \mathsf{add\text{-}arc}_G\ a &\coloneqq (\mathsf{V}_G \cup \{a_{\mathrm{t},G}, a_{\mathrm{h},G}\}, \mathsf{A}_G \cup \{a\}, \mathsf{tail}\ G, \mathsf{head}\ G) \\ G - \{a\} &\coloneqq (\mathsf{V}_G, \mathsf{A}_G \setminus \{a\}, \mathsf{tail}\ G, \mathsf{head}\ G) \\ \mathsf{add\text{-}vert}_G\ v &\coloneqq (\mathsf{V}_G \cup \{v\}, \mathsf{A}_G, \mathsf{tail}\ G, \mathsf{head}\ G) \\ G - \{v\} &\coloneqq (\mathsf{V}_G \setminus \{v\}, \mathsf{A}_G, \mathsf{tail}\ G, \mathsf{head}\ G) \end{aligned}$$

These functions are defined within pre-digraph, as they relate to a single graph. Functions taking more than one graph are defined outside of the pre-digraph locale.

The union of two digraphs is the digraph which has the arcs and vertices of both digraphs.

Definition 3.12 (Union)**.**

$$\begin{aligned} \mathsf{union}\ G\ H &\coloneqq (\mathsf{V}_G \cup \mathsf{V}_H, \mathsf{A}_G \cup \mathsf{A}_H, \mathsf{tail}\ G, \mathsf{head}\ G) \\ \mathsf{Union}_G\ \mathcal{G} &\coloneqq \left(\bigcup_{G \in \mathcal{G}} \mathsf{V}_G, \bigcup_{G \in \mathcal{G}} \mathsf{A}_G, \mathsf{tail}\ G, \mathsf{head}\ G \right) \end{aligned}$$

Note that for union, I arbitrarily choose the projection functions of G, so the definition yields useful results only if G and H are compatible. The union of a set of digraphs $\mathcal{G}$ is defined w.r.t. a graph G, so the result is always compatible to G, even if $\mathcal{G}$ is empty.

In [7], connected digraphs are defined in two steps, first referring to the underlying connected undirected graph, then referring to the associated strongly connected symmetric digraph. I leave out the intermediary and use the associated symmetric digraph to define symmetric reachability.

Definition 3.13 (Underlying Undirected Graph). *For $G : (\beta, \alpha)$* dg*, the underlying undirected graph of type β* pair-dg *is:*

$$\text{mk-symmetric } G \coloneqq (\mathsf{V}_G, \bigcup_{a \in \mathsf{A}_G} \{(a_{\mathrm{t},G}, a_{\mathrm{h},G}), (a_{\mathrm{h},G}, a_{\mathrm{t},G})\})$$

For the graph properties expressed as locales (cf. Figure 3.1), mk-symmetric G satisfies at least the same properties as G and is symmetric. Now, one can define connected digraphs.

Definition 3.14 (Connectivity and Strong Connectivity).

$$\begin{aligned}\text{strongly-connected } G &\coloneqq \mathsf{V}_G \neq \varnothing \wedge \forall u, v \in \mathsf{V}_G.\ u \to^* v\\ \text{connected } G &\coloneqq \text{strongly-connected (mk-symmetric } G)\end{aligned}$$

Note that I do not consider the null graph to be (strongly) connected. This makes the decomposition of digraphs into connected components unique.

Definition 3.15 (Subgraph and Strongly Connected Component). *A* subgraph *G of H is a well-formed graph G whose vertices and arcs are contained in H. If G contains all arcs of H connecting vertices of G, then G is an* induced subgraph.

$$\begin{aligned}\text{subgraph } G\ H &\coloneqq (\mathsf{V}_G \subseteq \mathsf{V}_H \wedge \mathsf{A}_G \subseteq \mathsf{A}_H \wedge\\ &\qquad \text{wf-digraph } G \wedge \text{wf-digraph } H \wedge \text{compatible } G\ H)\\ \text{induced-subgraph } G\ H &\coloneqq (\text{subgraph } G\ H \wedge \mathsf{A}_G = \{a \in \mathsf{A}_H.\ \{a_{\mathrm{t},H}, a_{\mathrm{h},H}\} \subseteq \mathsf{V}_H\}\end{aligned}$$

A subgraph G of H is maximal *for some property P, if it satisfies P and there is no other graph between G and H satisfying P.*

$$\begin{aligned}\text{max-subgraph } P\ G\ H &\coloneqq \text{subgraph } G\ H \wedge P\ G\\ &\qquad \wedge (\forall G'.\ \text{subgraph } G'\ H \wedge \text{subgraph } G\ G' \wedge P\ G' \implies G = G')\end{aligned}$$

A strongly connected component *(SCC) is a maximal strongly connected subgraph:*

$$\text{sccs } G \coloneqq \{H.\ \text{max-subgraph strongly-connected } H\ G\}$$

Induced subgraphs are also maximal subgraphs:

$$\text{induced-subgraph } G\ H = \text{max-subgraph } (\lambda G'.\ \mathsf{V}_{G'} = \mathsf{V}_G)\ G\ H$$

A classic result I proved for these properties is the unique decomposition of a symmetric digraph into strongly connected components:

Lemma 3.16. *For a symmetric digraph G, the decomposition into SCCs is unique:*

$$[\![S \subseteq \text{sccs } G;\ (\textstyle\bigcup_{H \in S} \mathsf{V}_H) = \mathsf{V}_G]\!] \implies S = \text{sccs } G$$

In many cases, one is only interested in the vertices of an SCC, not its full graph structure. Such sets of vertices have a nice characterization in terms of the reachability relation. As SCCs are either disjoint or equal, there is a bijection between SCCs and Vertex-SCCs.

Definition 3.17 (Vertex-SCCs). *The vertex-SCCs of a digraph G are* sccsV = verts (sccs G).

Lemma 3.18. *Let G be a wellformed digraph. Then a set of vertices S is a vertex-SCC of G if and only if S is a nonempty maximal set such that all elements of S are reachable from each other, that is:*

$$S \in \mathsf{sccsV}\ G \iff S \neq \varnothing \wedge (\forall u, v \in S.\ u \rightarrow^* v) \wedge (\forall u \in S.\ \forall v \notin S.\ \neg u \rightarrow^* v \vee \neg v \rightarrow^* u)$$

In informal language, I will refer to both SCCs and vertex-SCCS as SCCs. As vertex-SCCs can be characterized just by the reachability relation, without the arc or vertex set, they are often more convenient to use.

3.3.3. Digraph Isomorphisms

An isomorphism for digraphs consists of four functions: a mapping of the vertices, a mapping of the arcs and the new projection functions. For a given graph, the projection function could be derived from the vertex and arc mappings. However, as discussed below, providing explicit projection functions often eases reasoning.

Definition 3.19 (Digraph Isomorphism)**.**

record $(\beta, \alpha, \beta', \alpha')$ iso =
iso-verts : $\beta \to \beta'$, iso-arcs : $\alpha \to \alpha'$, iso-tail : $\alpha' \to \beta'$, iso-head : $\alpha' \to \beta'$

An isomorphism is applied to a digraph with the function

$$\mathsf{app\text{-}iso} : (\beta, \alpha, \beta', \alpha')\ \mathsf{iso} \to (\beta, \alpha)\ \mathsf{dg} \to (\beta', \alpha')\ \mathsf{dg}$$
$$\mathsf{app\text{-}iso}\ h\ G \coloneqq (\mathsf{iso\text{-}verts}\ h\ \mathsf{V}_G, \mathsf{iso\text{-}arcs}\ h\ \mathsf{A}_G, \mathsf{iso\text{-}tail}\ h, \mathsf{iso\text{-}head}\ h)$$

I write $h\,G \coloneqq$ app-iso $h\,G$, $h\,a \coloneqq$ iso-arcs $h\,a$, and $h\,v \coloneqq$ iso-verts $h\,v$. The predicate iso *expresses that h is an isomorphism between G and its image: It must be injective and preserve the graph structure.*

$$\mathsf{iso}_G\ h \coloneqq \mathsf{inj\text{-}on}\ h\ \mathsf{V}_G \wedge \mathsf{inj\text{-}on}\ h\ \mathsf{A}_G \wedge \left(\forall a \in \mathsf{A}_G.\ h\ a_{\mathrm{t},G} = (h\ a)_{\mathrm{t},h\,G} \wedge h\ a_{\mathrm{h},G} = (h\ a)_{\mathrm{h},h\,G}\right)$$

Two digraphs G, H are isomorphic, if there is an isomorphism mapping G to H:

$$\mathsf{digraph\text{-}iso}\ G\ H \coloneqq \exists h.\ \mathsf{iso}_G\ h \wedge \mathsf{app\text{-}iso}\ h\ G = H$$

An isomorphism h only needs to preserve the structure on A_G, not on the universe of α. That is, the behavior of iso-tail h and iso-head h outside of $\mathsf{A}_{h\,G}$ is arbitrary. This is in accordance with usual mathematic notation, but has the side-effect that projection functions of the image are defined uniquely only on a subset of α'. However, it is often possible to give sensible projections for the whole universe (for example fst and snd for pair digraphs) and using those eases reasoning.

I provide rewrite rules to reduce the basic selectors and many other functions on $h\,G$ to functions on G. For many predicates introduction rules are provided.

An inverse of an isomorphism can be determined automatically. Again, the inverse is only unique modulo the behavior of iso-tail and iso-head outside of A_G.

Definition 3.20 (Inverse Isomorphism)**.**

$$\text{inv-iso}_G\ h \coloneqq \Big((\text{iso-verts}\ h)^{-1}_{\text{V}_G}, (\text{iso-arcs}\ h)^{-1}_{\text{A}_G}, \text{head}\ G, \text{tail}\ G\Big)$$

For f injective on S, f^{-1}_S denotes the inverse function of f on S (defined by the choice operator).

The inverse isomorphism is useful to define a set of automatically usable simplification rules. As an example, consider the function in-deg returning the number of incoming arcs for a vertex. For an isomorphism h, the following two (conditional) equations reduce $\text{in-deg}_{h\,G}$ to in-deg_G:

$$u \in \text{V}_G \implies \text{in-deg}_{h\,G}\ (\text{iso-verts}\ h\ u) = \text{in-deg}_G\ u \tag{3.2}$$

$$[\![u \in \text{V}_{h\,G};\ u = \text{iso-verts}\ h\ v]\!] \implies \text{in-deg}_{h\,G}\ u = \text{in-deg}_G\ v \tag{3.3}$$

As usual in rewriting, Isabelle's simplifier can apply an equation if the left hand side matches a subterm of the goal and it is able to discharge the assumptions. Fresh variables on the right hand side are not allowed. So rule (3.2) can only be applied if the vertex is given in a very special form. The more general rule (3.3) is not suitable for the simplifier as the variable v does not occur on the left hand side. The function inv-iso allows us to get rid of this variable:

$$[\![u \in \text{V}_{h\,G}]\!] \implies \text{in-deg}_{h\,G}\ u = \text{in-deg}_G\ (\text{inv-iso}\ h\ u)$$

In particular, in combination with the other simplification rules I define for isomorphisms, this rule can be used whenever the simplifier can prove $u = \text{iso-verts}\ h\ v$ for some v. In this case, it rewrites to the same right hand side as (3.2). Some rewrite rules using this technique are shown below.

Many properties are preserved under isomorphisms, for example walks, trails and paths, and the reachability relation, but also the property of being reachable or a (maximal) subgraph.

Lemma 3.21. *Let h be a digraph isomorphism and let $u, v \in \text{V}_{h\,G}$. Let* set $p \subseteq \text{A}_{h\,G}$ *and P be a property on digraphs. I write h^{-1} for* $\text{inv-iso}_G\ h$*. Then*

$$\begin{aligned}
\text{awalk}_{h\,G}\ u\ p\ v &\iff \text{awalk}_G\ (h^{-1}\ u)\ (\text{map}\ h^{-1}\ p)\ (h^{-1}\ v)\\
\text{atrail}_{h\,G}\ u\ p\ v &\iff \text{atrail}_G\ (h^{-1}\ u)\ (\text{map}\ h^{-1}\ p)\ (h^{-1}\ v)\\
\text{apath}_{h\,G}\ u\ p\ v &\iff \text{apath}_G\ (h^{-1}\ u)\ (\text{map}\ h^{-1}\ p)\ (h^{-1}\ v)\\
u \rightarrow^*_{h\,G} v &\iff h^{-1}\ u \rightarrow^*\ h^{-1}\ v.\\
\text{subgraph}\ G\ H &\implies \text{subgraph}\ (h\ G)\ (h\ H)\\
\text{max-subgraph}\ P\ G\ H &\implies \text{max-subgraph}\ (P \circ h)\ (h\ G)\ (h\ H)
\end{aligned}$$

A *graph invariant* is a property P of digraphs which is invariant under isomorphism, that is $P\ G \iff P\ (h\ G)$ for all digraphs G and isomorphisms h. Obvious graph invariants are the number of arcs and the number of vertices. From the results above follows easily that also the number of isolated vertices (i.e., the vertices not incident to any arc) and the number of strongly connected components are invariants. In Section 4.4, I will show that planarity is a graph invariant.

3.4. Euler Graphs

In this section, I present a characterization of directed Euler graphs. The undirected variant of this theorem is one of the basic results any introductory textbook covers. I demonstrate that my library allows for a nice proof. An *Euler trail* is a trail that contains each arc of a graph exactly once and touches every vertex. Such a trail is *closed* if the two end vertices are equal and *open* otherwise. A digraph that has an Euler trail is called an *Euler digraph.*

Definition 3.22 (Euler Trail)**.**

$$\text{euler-trail}_G\ u\ p\ v \coloneqq \text{atrail}_G\ u\ p\ v \wedge \text{set}\ p = \mathsf{A}_G \wedge \text{set}\ (\text{aw-verts}_G\ u\ p) = \mathsf{V}_G$$

For connected graphs, the condition on the vertices can be dropped:

Lemma 3.23. *Let G be a connected digraph. Then the following equation holds:*

$$\text{euler-trail}_G\ u\ p\ v = (\text{atrail}_G\ u\ p\ v \wedge \text{set}\ p = \mathsf{A}_G)$$

I prove the following well-known characterization of Euler digraphs: a finite digraph has an Euler trail if and only if it is connected and either

- for all vertexes the in-degree equals the out-degree or
- there are two vertexes u and v such that the difference between in- and out-degree of u resp. v is -1 resp. 1 and for all other vertexes, the in-degree equals the out-degree.

In the first case, we get a closed Euler trail, in the second case an open Euler trail from u to v. The predicate arc-balanced defined below allows us to shorten the degree condition to $\exists u, v.\ \text{arc-balanced}_G\ u\ \mathsf{A}_G\ v$. The functions in-arcs, out-arcs, in-deg and out-deg denote the set of incoming/outgoing arcs of a vertex, respectively its cardinality.

Definition 3.24 (Arc Balance)**.**

$$\begin{aligned}\text{arc-balance}_G\ w\ A &= |\text{in-arcs}_G\ w \cap A| - |\text{out-arcs}_G\ w \cap A| \\ \text{arc-balanced}_G\ u\ A\ v &= \Big(\text{if } u = v \text{ then } \forall w \in \mathsf{V}_G.\ \text{arc-balance}_G\ w\ A = 0 \\ &\qquad \text{else } (\forall w \in \mathsf{V}_G \setminus \{u, v\}.\ \text{arc-balance}_G\ w\ A = 0) \wedge \\ &\qquad\quad \text{arc-balance}_G\ u\ A = -1 \wedge \text{arc-balance}_G\ v\ A = 1\Big)\end{aligned}$$

I proved that the conditions given above are necessary and sufficient. Here, I will only discuss the latter, that is, given the above conditions, an Euler trail exists. Let us first consider closed Euler trails.

Theorem 3.25 (Closed Euler Trail)**.** *Let G be a finite and connected digraph. Then:*

$$[\![\forall u \in \mathsf{V}_G.\ \text{in-deg}_G\ u = \text{out-deg}_G\ u]\!] \implies \exists u, p.\ \text{euler-trail}_G\ u\ p\ u$$

Proof. Note that the degree condition of this lemma is equivalent to

$$\forall u \in \mathsf{V}_G.\ \text{arc-balance}_G\ u\ \mathsf{A}_G = 0 \tag{3.4}$$

The proof given by Bang-Jensen and Gutin [7] is constructive: it starts with an empty trail and proves that a trail can always be extended until it contains all arcs.[2] Obviously an empty trail exists (as the empty digraph is not connected, G has at least one vertex). The extensibility argument is inductive; I model this as an induction on the number of arcs not in the trail. For the induction step, note that trails are always balanced, i.e.

$$[\![\text{trail}_G\ u\ p\ v]\!] \implies \text{arc-balanced}_G\ u\ p\ v \tag{3.5}$$

We need to consider two cases. If the end vertices of the trail are distinct, (3.4) and (3.5) guarantee that an arc incident to one of the endpoints exists.

If the trail is closed, we need to find an arc that "touches" the trail (i.e., has a common vertex with the trail, but is not part of it). Then we rotate the trail, such that the common vertex is at both ends and attach the arc to one of the ends. Such an arc exists as the digraph is connected: if an arc does not touch the trail, then it is incident to a vertex which is not in the trail. As the digraph is connected, there is a walk from the start of the trail to this vertex. Then Lemma 3.10 can be used to derive a contradiction.

So far, we have proved that an Euler trail p exists. For p holds set $p = \mathsf{A}_G$ and by (3.5) follows $\text{arc-balanced}_g\ u\ \mathsf{A}_G\ v$. By (3.4) and the definition of arc-balanced this walk is closed. □

The textbook proof of the characterization of open Euler trails usually proceeds as follows: Let u, v be the vertices with $\text{arc-balance}_G\ u\ \mathsf{A}_G = -1$ and $\text{arc-balance}_G\ u\ \mathsf{A}_G = 1$. Add an arc from v to u to G. The resulting graph has a closed Euler trail. Removing the additional arc from this trail yields an open Euler trail for G.

With the usual set-theoretic formulation, it is always possible to add an additional arc between two vertices to a digraph. However, in the type system of Isabelle/HOL the universe of the arc type might already be exhausted by the existing arcs. A pathological example for this case was given in Section 3.2, but this issue is not restricted to our abstract graph representation; more concrete representations, like $\alpha = \beta \times \gamma \times \beta$, are affected, too, if only because one needs to prove that γ cannot be exhausted.

This problem can be avoided by requiring γ to be infinite. As we are only interested in trails of finite digraphs, this ensures that γ is not exhausted. From this result for a specialized graph type, the result for arbitrary arc types α can be proven by isomorphism.

Lemma 3.26. *Let* $G : (\beta, \beta \times \mathbb{N} \times \beta)\ \mathsf{dg}$ *be a finite, connected digraph with* $\mathsf{tail}\, G = \mathsf{fst}$, $\mathsf{head}\, G = \mathsf{snd} \circ \mathsf{snd}$ *and natural numbers as arc labels. Then*

$$[\![\{u, v\} \subseteq \mathsf{V}_G;\ \text{deg-cond}\ G\ u\ v]\!] \implies \exists p.\ \text{euler-trail}_G\ u\ p\ v$$

holds, where deg-cond $G\ u\ v$ *abbreviates the following condition:*

$$(\forall w \in \mathsf{V}_G \setminus \{u, v\}.\ \text{in-deg}_G\ w = \text{out-deg}_G\ w) \wedge$$
$$\text{in-deg}_G\ u + 1 = \text{out-deg}_G\ u \wedge \text{out-deg}_G\ v + 1 = \text{in-deg}_G\ u$$

[2] Bang-Jensen and Gutin [7] restrict their proof to loop-free digraphs. This is not necessary as loops do not need special handling in this proof.

Proof. We construct a new graph $H \coloneqq \text{add-arc}_G\ (v, \ell, u)$ where $\ell \in \mathbb{N}$ is a label not occurring in G. Such a label exists as $\mathbb{N}$ is infinite. H is again a finite digraph. For the interpretation of fin-digraph with H we use rewriting equations similar to those in Section 3.2.1.

H satisfies the degree condition of Theorem 3.25. After a case analysis with the cases $u = w$, $u \neq w$, $v = w$ and $v \neq w$, we can automatically prove this for all $w \in \mathsf{V}_H$ using the obvious lemmas relating add-arc and degrees and obtain a closed Euler trail for H. This Euler trail contains the arc (v, ℓ, u). We rotate the trail so that this arc is the first arc of the trail. Removing this arc then yields an open Euler trail for G. □

From this, one can prove the version for arbitrary arc types. I use the results about isomorphisms from Section 3.3.3 to transfer Lemma 3.26.

Theorem 3.27 (Open Euler Trail). *Let* $G : (\beta, \alpha)\ \mathsf{dg}$ *be a finite and connected digraph.*

$$[\![\{u, v\} \subseteq \mathsf{V}_G;\ \text{deg-cond}\ G\ u\ v]\!] \implies \exists p.\ \text{euler-trail}_G\ u\ p\ v$$

Proof. As G is finite, there is function $f : \alpha \to \mathbb{N}$ which is injective on A_G. From f, we construct an isomorphism h:

$$\mathsf{h} = (\lambda v.\ v, \lambda a.\ (\text{tail}\ G\ a, f\ a, \text{head}\ G\ a), \text{fst}, \text{snd} \circ \text{snd})$$

The property $\text{iso}_G\ \mathsf{h}$ follows by unfolding the definitions of iso and h and the injectivity of f. We then obtain an Euler trail for the graph $\mathsf{h}\ G$ by using Lemma 3.26. Using the rewrite rules for isomorphism the assumptions of the lemma can be discharged automatically.

We transfer the trail to $G = (\text{inv-iso}\ \mathsf{h})\ (h\ G)$ by applying the inverse isomorphism. The proof is again a simple application of the isomorphism lemmas. □

3.5. Other Graph Formalizations

In the journal version of this article [60], I wrote I "expect [the graph library] to become the standard for formalizations about directed graphs in Isabelle". Indeed, the library is being used in other projects [2, 27, 69]. On the other hand, there is recent work where the authors chose to use their own graph formalization, although they were aware of this formalization. In this section, I try to provide some insights why this is the case. Related to this is the question how to integrate results proven using different formalizations of graphs.

It takes a certain amount of time to become familiar with a new library, even if one is familiar with theorem proving and the topic at hand. One not only needs to learn the vocabulary, but also how to state theorems and intermediate steps so that the provided automation can work efficiently. For many areas of mathematics, it is immediately obvious that it is easier to familiarize oneself with the existing theories than to start from scratch.

As an example, consider the real numbers: a typical user knows how to work with them, but not how to define them. Also, after defining real numbers, one wants to forget about the definition in terms of Dedekind cuts or Cauchy sequences and work with the abstract objects. To prove any interesting results, one already needs a large body of elementary theorems, abstracting from the raw definition.

On the other hand, graph theory often seems simple enough to not bother with an existing library. In many cases, there is no abstraction from the tuple-of-sets representation and basic algorithms like Dijkstra's algorithm or the Ford-Fulkerson algorithm can be considered without deep insights in graph theory. Many results have quite elementary proofs, so formalizing them is a feasible task, even without an existing formalization of graph theory.

Another motivation is to keep the representation of graphs as simple as possible. While this is often useful (which was also the motivation for introducing the pair digraphs in Section 3.2.1), people tend to overestimate the benefits and the work necessary for deriving a simpler representation from the graph library. More important, it makes it harder to reuse their results, because each result has its own vocabulary.

I will discuss this on the example of a recent formalization of the max-flow/min-cut theorem by a student. After some discussion, he and his supervisor decided to use their own graph definition. One reason for this was that, in their context, a digraph is given simply by a cost function $\beta \to \mathbb{N}$, so they wanted to keep the definition simple. Their initial theory started with less then a screen full of definitions related to basic graph theory. Of the 3900 lines of proof text of their final result, around 15% prove results already available in the graph library, and in the end they defined arc and vertex sets explicitly. If one had used the graph library to define a digraph from the cost function then most of those lemmas could have been derived by a one-line proof (and my guess is that this would not even have been necessary in many cases). So while the cost of a new formalization of graphs was not prohibitive, this project would have profited from using my graph library.

The weighting of costs and benefits may change if the difference between graph representations is larger: for example, in Chapter 2 I presented a formalization of the Girth-Chromatic Theorem. Being a mostly combinatorial result on undirected graphs, it would have been harder to prove this result in the context of the graph library.

In both cases, using a separate formalization of graphs hurts the possibility to combine these results, so they should be phrased in terms of a common graph representation. For the max-flow-min-cut theorem, it would probably be possible to just exchange the basic definition and fix the proofs that break. In general, this is not a feasible approach.

Thanks to the work of Huffman and Kunčar [38], Isabelle has tools to transfer theorems and lift definitions between a type and a subtype. For example, undirected graphs are in bijection to bidirected graphs and hence can be considered as a subtype of directed graphs. Then, by proving a *transfer theorem* for each constant, theorems for undirected graphs can be automatically transformed into theorems about bidirected digraphs. This allowed me to transfer main result of Chapter 2 into the graph library with just about 230 lines of proof text:

Lemma 3.28. *Let ℓ be a natural number. Then there is a symmetric and loop-free digraph G : $\mathbb{N}$ pair-dg, such that $\ell <$ g′ G and $\ell < \chi'$ G.*

Here, g′ and χ' are the girth and the chromatic number, defined on digraphs. This result is restricted to vertices of type $\mathbb{N}$ and arcs of type $\mathbb{N} \times \mathbb{N}$, as this corresponds to the definition of undirected graphs in Chapter 2. It is possible to generalize this result to the type (β, α) dg for infinite types β and α by giving an isomorphism, similar to the approach used in Section 3.4.

The definition of graphs given in this chapter is very general. This leads me to conclude that the graph library presented in this chapter is a good basis to collect results about graph theory,

even if they have been proven using another definition of graphs.

3.6. Conclusion

In this chapter, I presented the first implementation of a graph library in Isabelle/HOL covering a wide range of fundamental properties. Based on that, I formalized a characterization of Euler digraphs. Some results on planarity I formalized using this library are found in the following chapters. This library is used in other projects [2, 27, 69], so it useful outside the topics of this thesis.

To achieve the goal of an universal library, it is important that the chosen graph representation can be used to express as much of directed graph theory as possible. Therefore I chose an abstract representation that supports multi-digraphs and can be instantiated to many common representations, stemming from both mathematics (arcs as pairs of vertices) and implementation (arcs are pointers into some data structure).

To make proofs convenient, I provided a careful setup of automation mechanisms. The heuristics are able to discharge a large number of frequently occurring goals. Examples for the setup are the special simplification procedure from Section 3.3.1 and the use of locales to recover less general graph representations, but also the introduction of the inverse isomorphism.

The formalization consists of around 6500 lines of Isabelle theories. The Euler case study builds on this and needs 670 lines of proof. For comparison, the Mizar characterization of undirected Euler Graphs[56] consists of 3500 lines, based on 5500 lines of graph theory.

4. Planarity of Graphs

A graph is planar if it can be drawn on a plane (or on a sphere) such that there are no intersecting edges. This intuitive definition is hard to use in proofs. In this chapter, I formalize two well-known alternative characterizations of planarity: one is a combinatorial characterization, based on Euler's polyhedral formula, the other one is based on Kuratowski's theorem. I implement an executable Isabelle function deciding combinatorial planarity (for small graphs) and prove it to be correct. This decision procedure is then used to prove that the two characterizations are compatible in the sense that each graph which is "combinatorially planar" is also "Kuratowski-planar".

All results in this chapter have been proven in Isabelle and build upon the graph library described in Chapter 3.

A small theory of permutations in Section 4.1 lays the basis for the combinatorial characterization of planarity. Section 4.2 formally introduces two definitions of planarity. In Section 4.3, I give an executable specification of combinatorial planarity and prove that combinatorial planarity is preserved under subdivisions and subgraphs in Section 4.5 and Section 4.6, respectively. This leads to the compatibility result. The chapter concludes with a discussion in Section 4.7.

4.1. Permutations

A function $f : \alpha \to \alpha$ is a *permutation*, if f is bijective and there is a finite set S such that $f\,x = x$ for all $x \notin S$. In this case, we also say that f *permutes* S. A permutation f with $f\,(f\,x) = x$ for all x is an *involution*. In the remainder of this section f is always a permutation.

Permutations are closed under composition, i.e., if f permutes S and g permutes T, then $f \circ g$ permutes $S \cup T$.

The *orbit* of x under f is the set of all elements reachable from x by applying f repeatedly. This is inductively defined as the smallest set such that

$$f\,x \in \text{orbit}\,f\,x \qquad\qquad y \in \text{orbit}\,f\,x \implies f\,y \in \text{orbit}\,f\,x$$

hold. This is equivalent to

$$\text{orbit}\,f\,x = \{f^n\,x \mid 0 \le n\} = \{f^n\,x \mid 0 < n\}.$$

The relation "x is in the orbit of y" is an equivalence relation. In particular are orbit $f\,x$ and orbit $f\,y$ either equal or disjoint. Recall that we lift functions to sets implicitly, so for some set X, orbit $f\,X = \{\text{orbit}\,f\,x \mid x \in X\}$.

A *segment* is a subset of the orbit: segment $f\,x\,y$ contains the elements reachable from x by repeated application of f until y is reached. This is formally defined as the smallest set

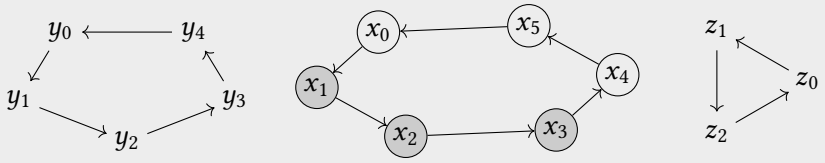


Figure 4.1.: Comparison of orbit $f\ x_0$ versus segment $f\ x_0\ x_4$ where f is a permutation on $\{x_0, \ldots, x_5, y_0, \ldots, y_4, z_0, \ldots, z_2\}$. The circled nodes are the elements of orbit $f\ x_0$, the darker nodes are the elements of segment $f\ x_0\ x_4$. Each arrow represents a single application of f.

satisfying the following two implications:

$$f\ x \neq y \implies f\ x \in \text{segment}\ f\ x\ y$$
$$[\![x \in \text{segment}\ f\ x\ y; f\ x \neq y]\!] \implies f\ x \in \text{segment}\ f\ x\ y$$

An equivalent definition is segment $f\ x\ y = \{f^n\ x \mid 0 < n \wedge n < k\}$ where k is the smallest number greater 0 such that $f^k\ x = y$. See Figure 4.1 for an example. In particular, the equation segment $f\ x\ x = \text{orbit}\ f\ x \setminus \{x\}$ holds .

A permutation f is cyclic on a set S, if S is an orbit of f, i.e.,

$$\begin{aligned}\text{cyclic-on}\ f\ S \coloneqq \quad & \exists s \in S.\ \text{orbit}\ f\ s = S \\ \iff & S \neq \varnothing \wedge (\forall s \in S.\ \text{orbit}\ f\ s = S)\end{aligned}$$

If f permutes S and is cyclic on S, we also say that f is cyclic. Permutations can be represented as products of cyclic permutations, i.e., for each permutation f there are cyclic permutations $f_1, \ldots, f_n$ and sets $S_1, \ldots, S_n$ such that $f = f_n \circ \cdots \circ f_1$, f_i permutes S_i and $S_i \cap S_j = \varnothing$ for $i \neq j$. Cycles and products of disjoint cycles can naturally be represented by lists and lists of lists, respectively.

Definition 4.1 (Cycles). *We say that xs is a* cycle *if and only if xs is a list of distinct elements. This property is expressed by the predicate* distinct*. We say that xss is a* product of disjoint cycles *if and only if xss is a list of non-empty cycles and no two cycles in xss have a common element. Formally, we define the predicate* distincts *to express the property "product of disjoint cycles":*

$$\begin{aligned}\text{distincts}\ xss \coloneqq\ & \text{distinct}\ xss \wedge (\forall xs \in xss.\ \text{distinct}\ xs \wedge xs \neq [\,]) \\ & \wedge (\forall xs, ys \in xss.\ xs \neq ys \implies \text{set}\ xs \cap \text{set}\ ys = \varnothing)\end{aligned}$$

The function sset $xss \coloneqq \text{set}\ (\text{map set}\ xss)$ *gives the orbits described by a product of disjoint cycles.*

Let $xs = [x_0, \ldots, x_{n-1}]$ *be a cycle. Then, the permutation represented by this list is defined as*

$$\text{cycle}\ xs\ x \coloneqq \begin{cases} x_{(i+1)\bmod n} & \text{if } x = x_i \text{ for some } 0 \leq i < n \\ x & \text{else.}\end{cases}$$

The permutation represented by a product of disjoint cycles is defined by the following equations:

$$\text{cycles}\ [\,]\ x \coloneqq x \qquad \text{cycles}\ (xs :: xss) \coloneqq \text{cycle}\ xs\ (\text{cycles}\ xss\ x)$$

Lemma 4.2. *Let xs be a non-empty cycle and xss a product of disjoint cycles. Then,* cycle *xs is cyclic on and permutes* set *xs. Similarly,* cycles *xss is cyclic on each of the elements of* sset *xss and permutes* $\bigcup$ sset *xss.*

Rotation of cycles does not change the represented permutation. That is, for a cycle xs with $xs = ys +\!\!+ zs$, the cycle $zs +\!\!+ ys$ yields the same permutation. Note that cycle $xs\ x \in xs \iff x \in xs$. A similar property holds for cycles. For the work in this chapter, (products of disjoint) cycles have a useful property: the explicit representation as lists makes them easy to enumerate and manipulate in executable code. In particular, the two operations perm-swap and perm-rem defined below have a straightforward equivalent in terms of common list operations.

Permutations can be restricted to a specific domain by

$$\text{perm-restrict}\ f\ S\ x \coloneqq \begin{cases} f\ x & \text{if } x \in S \\ x & \text{else.} \end{cases}$$

This is again a permutation, if and only if S is a union of orbits of f, that is $S = \bigcup \text{orbit}\ f\ S$. In particular, the following properties hold:

Lemma 4.3. perm-restrict f (orbit $f\ x$) *is a cyclic permutation.*

Lemma 4.4. *If f permutes S and f is cyclic on A, then* perm-restrict f $(S \setminus A)$ *permutes $S \setminus A$.*

Lemma 4.5. *If* perm-restrict $f\ A$ *and* perm-restrict $f\ B$ *are permutations and $A \cap B = \varnothing$, then*

$$\text{perm-restrict}\ f\ A \circ \text{perm-restrict}\ f\ B = \text{perm-restrict}\ f\ (A \cup B).$$

I conclude this section by introducing two further operations on permutations.

Definition 4.6 (Modifying Permutations)**.** *Let $f : S \to S$ be a permutation and $x, y \in S$. Then:*

$$\text{perm-swap}\ x\ y\ f \coloneqq (x \rightleftharpoons y) \circ f \circ (x \rightleftharpoons y) \quad \text{perm-rem}\ x\ f \coloneqq \begin{cases} (x \rightleftharpoons f\ x) \circ f & \text{if } f\ x \neq x \\ f & \text{else} \end{cases}$$

where $x \rightleftharpoons y$ is the permutation swapping x and y. For cycles, this corresponds to:

$$\text{perm-rem}\ x\ (\text{cycle}\ xs) = \text{cycle}\ (xs - x)$$
$$\text{perm-swap}\ x\ y\ (\text{cycle}\ xs) = \text{cycle}\ (\text{map}\ (\text{cycle}\ [x, y])\ xs)$$

Example 4.7. *Let f =* cycles $[[1, 2, 3], [4, 5]]$ *be a permutation. Then the following equations hold:*

$$\text{perm-rem}\ 1\ f = \text{cycles}\ [[2, 3], [4, 5]] \qquad \text{perm-rem}\ 4\ f = \text{cycles}\ [[1, 2, 3], [5]]$$
$$\text{perm-swap}\ 1\ 3\ f = \text{cycles}\ [[3, 2, 1], [4, 5]] \qquad \text{perm-swap}\ 2\ 4\ f = \text{cycles}\ [[1, 4, 3], [2, 5]]$$

If f permutes S, then perm-rem $x\ f$ permutes the set $S \setminus \{x\}$ and perm-swap $x\ y\ f$ permutes cycle $[x, y]$ S.

4.2. Two Definitions of Planarity

In this section, I formalize two well-known characterizations of planarity.

Combinatorial Planarity I start with a brief excursion to topology to motivate the definition of combinatorial planarity given below. For a detailed discussion, see for example Lando and Zvonkin [44]. An undirected graph G with vertices V and edges E is planar, if it can be drawn on a plane (or sphere) without intersecting edges. More general, a topological map is a drawing of a connected graph on a surface without intersecting edges, satisfying the following property: if the surface is cut at the drawn edges, then each of the remaining faces is homeomorphic to an open disc. For a map, the equation

$$|V| - |E| + |F| = 2 - 2g$$

holds, where F is the set of faces and g the genus of the surface. Intuitively, the genus of a surface is the number of holes in it: a surface with genus 0 is homeomorphic to a sphere, a surface with genus 1 is homeomorphic to a torus and so on. The genus of a map is the genus of the underlying surface. Hence, a graph is planar if there is a plane map, i.e., a map with genus 0. Then the formula above is Euler's polyhedral formula.

A graph with multiple components is planar, if each of the components is planar, or equivalently, if each component has a planar map. If g refers to the sum of the genera of the maps, this leads to the generalized equation

$$|V| - |E| + |F| = 2k - 2g,$$

where k is the number of components of the graph.

In a map, an edge is always the border between two faces. For combinatorial maps, one splits the (undirected) edges into two (directed) arcs per edge[1] and assigns an arc to the face on its left hand side. Each face can now be represented by a cyclic permutation enumerating the arcs of a face counter-clockwise. The product of all cyclic permutations is the *face cycle successor* function σ. There are also two other permutations which can be defined on the arcs: the involution π, mapping an arc to the other arc belonging to the same edge, and the permutation ρ, enumerating the outgoing arcs of a vertex clockwise. I also call the former *arc reversal* permutation and the latter *arc rotation* permutation. Note that two of these permutations uniquely determine the third, in particular the equation $\sigma = \rho \circ \pi$ holds.[2]

The face cycle successor function σ partitions the arcs of the graph G according to the faces they belong to. A map for an *isolated vertex* (i.e., a vertex without outgoing arcs) always has exactly one face without any arcs, so we get $F = |\text{orbit } \sigma\ A| + |\text{isolated-verts } G|$, where isolated-verts G is the set of isolated vertices and A the set of arcs. This leads to the following characterization of planarity for arbitrary finite graphs [52]:

$$V - E + |\text{orbit } \sigma\ A| + |\text{isolated-verts } G| = 2k - 2g$$

[1] often called half-edges or darts in the literature

[2] Lando and Zvonkin [44] consider a different direction for the face cycles, leading to the equation $\rho \circ \pi \circ \sigma = \text{id}$. Inverting the direction makes the equations slightly simpler for the purposes of this chapter. The symmetric definition by Lando and Zvonkin [44] leads to a more general concept of hypermaps (by dropping the restriction $\pi^2 = \text{id}$).

This motivates the following definition of a combinatorial map for graph. The definition is based on bidirected digraphs, not on undirected graphs. This is preferable as all operations are defined on arcs, not on edges.

Definition 4.8 (Combinatorial Map, Face Cycles). *A* combinatorial map *(or just* map*) is a tuple* $M = (G, \pi, \rho)$ *such that* (G, π) *is a bidirected graph and* ρ *permutes* A_G *such that all arcs with the same tail are in one orbit. Formally, the latter property is defined by*

$$\text{rotates}_G\ \rho \coloneqq (\forall v \in \mathsf{V}_G.\ \text{out-arcs}\ G\ v \neq \varnothing \implies \text{cyclic-on}\ \rho\ (\text{out-arcs}\ G\ v))$$

and the predicate digraph-map M *expresses that* M *is a map.*

For such a combinatorial map, σ_M *is defined as* $\sigma_M = \rho \circ \pi$*. Then, the* face cycles *of this map are the orbits of* σ_M*:*

$$\mathsf{F}\ M \coloneqq \text{orbit}\ \sigma_M\ \mathsf{A}_G$$

For a biconnected graph (G, π), we use the term edge to refer to both a and $\pi\ a$ for some arc $a \in \mathsf{A}_G$. If the map $M = (G, \pi, \rho)$ is clear from the context, we write G respectively σ_G instead of M and σ_M.

Definition 4.9 (Genus of a Map, Combinatorial Planarity). *Let* $M = (G, \pi, \rho)$ *be a combinatorial map. Then the* genus *of this map is defined as*

$$\text{genus}\ M \coloneqq \lfloor (2 \cdot |\text{sccs}\ G| - |\mathsf{V}_G| + |\mathsf{A}_G|/2 - |\mathsf{F}\ M| - |\text{isolated-verts}\ G|)/2 \rfloor$$

A map M *is called* plane*, if* genus $M = 0$*. A graph* G *is* combinatorially planar *if there is a plane map for this graph, i.e.,*

$$\text{comb-planar}\ G \coloneqq \exists \pi, \rho.\ \text{digraph-map}\ (G, \pi, \rho) \wedge \text{genus}\ (G, \pi, \rho) = 0$$

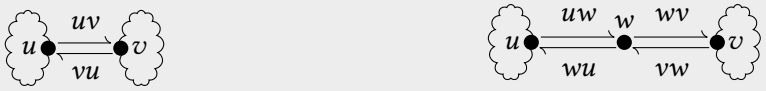
The rounding in the definition of genus is justified as the dividend is even [52].

Kuratowski planarity Another characterization of planarity is due to Kuratowski. A graph H is a *subdivision* of a graph G, iff H can be derived from G by repeatedly inserting vertices into edges. Let K_5 be the complete graph on five vertices and $K_{3,3}$ the complete bipartite graph on three and three vertices. We call $K_{3,3}$ and K_5 *Kuratowski graphs.* Kuratowski's theorem characterizes planarity.

Theorem 4.10 (Kuratowski [43]). *A graph* K *is a* Kuratowski subgraph *of* G *if* K *is a subgraph of* G *and the subdivision of a Kuratowski graph. A graph* G *is planar if and only* G *has no Kuratowski subgraph.*

Formally, complete (bipartite) digraphs are defined as follows:

Definition 4.11 (Complete (Bipartite) Digraphs). *A digraph* G *is* complete *on* m *vertices,* K_m*, if it has exactly* m *vertices and all vertices are pairwise connected (without loops).* G *is a* complete bipartite *digraph on* m *and* n *vertices,* $\mathsf{K}_{m,n}$*, if it consists of two sets of* m *and* n *vertices, and*



(a) G (b) H

Figure 4.2.: Graphs G and H satisfying subdiv-step (G, π_G) (H, π_H) (u, v, w) (uv, uw, vw).

two vertices are connected if they are not in the same set. Formally, the predicates are defined as follows:

$$\mathsf{K}_n\ G \coloneqq \mathsf{graph}\ G \wedge |\mathsf{V}_G| = n \wedge \mathsf{adj}_G = \{(u, v) \mid (u, v) \in \mathsf{V}_G \times \mathsf{V}_G \wedge u \neq v\}$$
$$\mathsf{K}_{m,n}\ G \coloneqq \mathsf{graph}\ G \wedge (\exists U, V.\ \mathsf{V}_G = U \cup V \wedge U \cap V = \varnothing \wedge |U| = m \wedge |V| = n \wedge \mathsf{adj}_G = U \times V \cup V \times U)$$

A graph G satisfies the predicate graph*, if G is loop-free, symmetric and has no parallel arcs.*

As with combinatorial planarity, the definition is based on bidirected graphs, which allows us to reuse our existing development. A single step of "splitting an edge" is defined below. Figure 4.2 illustrates this definition.

Definition 4.12 (Subdivision Step). *A bidirected graph (H, π_H) is derived by a* subdivision step *from a bidirected graph (G, π_G) if H is derived from G by inserting a new vertex w into an edge $\{uv, \pi_G\ uv\}$, replacing that edge with the fragment $\{uw, \pi_G(uw)\}$ w $\{vw, \pi_G\ vw\}$. Formally, this is defined as follows:*

$$\begin{aligned}
&\mathsf{subdiv\text{-}step}\ (G, \pi_G)\ (H, \pi_H)\ (u, v, w)\ (uv, uw, vw) \coloneqq \\
&\quad \mathsf{bidirected\text{-}digraph}\ (G, \pi_G) \wedge \mathsf{bidirected\text{-}digraph}\ (H, \pi_H) \\
&\quad \wedge\ \mathsf{perm\text{-}restrict}\ \pi_H\ \mathsf{A}_G = \mathsf{perm\text{-}restrict}\ \pi_G\ \mathsf{A}_H \wedge \mathsf{compatible}\ G\ H \\
&\quad \wedge\ \mathsf{V}_H = \mathsf{V}_G \cup \{w\} \wedge w \notin \mathsf{V}_G \\
&\quad \wedge\ \mathsf{A}_H = \{uw, \pi_H\ uw, vw, \pi_H\ vw\} \cup \mathsf{A}_G - \{uv, \pi_G\ uv\} \\
&\quad \wedge\ uv \in \mathsf{A}_G \wedge \mathsf{distinct}\ [uw, \pi_H\ uw, vw, \pi_H\ vw] \\
&\quad \wedge\ uv_{\mathsf{t},G} = u \wedge uv_{\mathsf{h},G} = v \wedge uw_{\mathsf{t},G} = u \wedge uw_{\mathsf{h},G} = w \wedge vw_{\mathsf{t},G} = v \wedge vw_{\mathsf{h},G} = w
\end{aligned}$$

Definition 4.13 (Subdivision). *A bidirected graph (H, π_H) is a* subdivision *of a bidirected graph (G, π_G), if it can be derived from G by repeated subdivision steps. Formally, the* subdivision *predicate is defined inductively as the smallest set satisfying the following conditions:*

$$\begin{aligned}
&\mathsf{bidirected\text{-}digraph}\ (G, \pi_G) \\
\Longrightarrow\ &\mathsf{subdivision}\ (G, \pi_G)\ (G, \pi_G)
\end{aligned}$$

$$\begin{aligned}
&[\![\mathsf{subdivision}\ (G, \pi_G)\ (I, \pi_I); \mathsf{subdiv\text{-}step}\ (I, \pi_I)\ (H, \pi_H)\ (u, v, w)\ (uv, uw, vw)]\!] \\
\Longrightarrow\ &\mathsf{subdivision}\ (G, \pi_G)\ (H, \pi_H)
\end{aligned}$$

Following Kuratowski's theorem, we arrive at the following definition of planarity:

Definition 4.14 (Kuratowski Planarity).

$$\text{kuratowski-planar } G \coloneqq \neg(\exists(H, \pi_H), (K, \pi_K).\ \text{subgraph } H\ G \\ \wedge\ \text{subdivision } (K, \pi_K)\ (H, \pi_H) \wedge (\mathsf{K}_{3,3}\ K \vee \mathsf{K}_5\ K))$$

Note that these two characterizations of planarity – the combinatorial one and the one based on Kuratowski's theorem – are in some sense dual: a plane map proves planarity of a graph, while a Kuratowski subgraph proves non-planarity of a graph. The planarity test implemented in the LEDA library [52] uses both to provide a certificate for the result. A separate checker algorithm then uses these certificates to ensure that the result is correct. In Chapter 5 and Chapter 7, I will verify checkers for both the negative and the positive certificate. A successful run of the former guarantees that the graph is not Kuratowski-planar, a successful run of the latter guarantees combinatorial planarity. In the remainder of this chapter, I prove that combinatorial planarity implies Kuratowski planarity. This allows me to prove that a successful check of the negative certificate guarantees that the graph is not combinatorially planar, i.e., the correctness theorems for both checkers talk about the same definition of planarity.

4.3. An Executable Specification of Combinatorial Planarity

Consider the definition of combinatorial planarity. As there are finitely many maps, enumerating all possible maps and computing the genus leads to a simple decision procedure for this property. This decision procedure is inefficient (with a runtime far worse than exponential), but it suffices for deciding planarity of small graphs, for example of a $\mathsf{K}_{3,3}$ or K_5. In this section, I describe an implementation of this decision procedure in Isabelle, together with a formal proof of its correctness. This allows a user to decide planarity of small graphs with the use of the Isabelle's eval proof method. I then use this implementation to prove that the Kuratowski graphs are not combinatorially planar.

Isabelle's code generator can evaluate an expression if for all constants occurring in this expression a purely equational specification is given. In particular, the usual set operations can be evaluated, provided that the sets have the form set *xs* for some list *xs*. This includes quantification over some finite set. Hence, in this section I represent graphs by a pair of lists: a list of vertices *vs* and list of arcs *as*, where each arc is a pair of vertices. I write $\mathsf{LG}(vs, as)$ for the pair digraph (cf. Section 3.2.1) represented by (vs, as).

In Section 4.3.1, I present an executable enumeration of all maps. Similarly, Section 4.3.2 contains an executable characterization of the genus. In Section 4.3.2, I use these to prove the non-planarity of a concrete K_5 and $\mathsf{K}_{3,3}$.

4.3.1. Enumerating All Maps

How can we enumerate all maps for a graph? Recall that a map is a tuple (G, π, ρ), where π is an arc reversal and ρ an arc rotation. For the pair graphs used in this section, π is uniquely determined, so the problem can be reduced to enumerating all possible values for ρ.

Lemma 4.15. *Let G be a graph and* to-map $G\ \rho \coloneqq (G, \text{perm-restrict swap } \mathsf{A}_G, \rho)$, *where* swap $(x, y) \coloneqq (y, x)$. *Then the set of maps for G is given by*

$$\text{to-map } G\ \{f \mid f \text{ permutes } \mathsf{A}_G \wedge \text{rotates}_G\ f\}.$$

If (G, π, ρ) is a map, then ρ is a product of disjoint cycles, where each cycle consists of the outgoing arcs of one vertex of G. So, the arc rotation permutations can be enumerated by first enumerating all possible cycles for each vertex separately and then building all possible products of disjoint cycles which can be constructed from these cycles. The following lemmas tell us which products of disjoint cycles we need to consider.

Lemma 4.16. *Let f be a permutation of S. Let xss be a product of disjoint cycles, such that $S = \bigcup$ sset xss and for each $s \in S$ there is a $xs \in xss$ where*

$$\text{perm-restrict } f\ (\text{orbit } f\ s) = \text{cycle } xs.$$

Then $f =$ cycles xss.

Proof. Proof by induction on *xss* for all f, S satisfying the above conditions. For $xss = []$, f is the identity function and so is cycles []. For $xss = xs :: xss'$, we have

$$\text{perm-restrict } f\ (\text{orbit } f\ s) = \text{cycle } xs$$

for any $s \in xs$ and hence

$$\text{perm-restrict } f\ (\text{set } xs) = \text{cycle } xs.$$

In particular is perm-restrict f (set *xs*) cyclic on *xs*. By Lemma 4.4 we know that

$$\text{perm-restrict } f\ (\bigcup \text{sset } xss')$$

is a permutation on $\bigcup$ sset *xss'* and hence we can use the induction hypothesis to derive

$$\text{perm-restrict } f\ (\bigcup \text{sset } xss') = \text{cycles } xss'.$$

With Lemma 4.5, the following sequence of equations

$$f = \text{perm-restrict } f\ (xs \cup xss') = \text{cycle } xs \circ \text{cycles } xss' = \text{cycles } xss$$

holds. This concludes the proof. □

By the above lemma, a suitable product of disjoint cycles represents a permutation f. Assume that we the orbits of f are given as a list of lists *css*. The following lemma now tells us that a suitable product of disjoint cycles can be obtained by permuting each list in *css* appropriately. Rotating a cycle does not change the associated cyclic permutation, so the first element of each list can stay the same.

Corollary 4.17. *Let f be a permutation of S and let css be a product of disjoint cycles such that $S = \bigcup$ css. Moreover, assume that for each cycle $cs \in css$,* set *cs is an orbit of f.*

Then there is a product of disjoint cycles xss with $f =$ cycles *xss such that* map set *xss =* map set *css and* map hd *xss =* map hd *css.*

Proof. Here *css* is a list of orbits. Let $cs \in css$. Then perm-restrict f (sset cs) is a cyclic permutation and there is a cycle xs such that cycle xs = perm-restrict f (sset cs). We have set xs = set cs. As rotating a cycle yields the same permutation, xs can be chosen to start with the same element as cs. The theorem then follows from Lemma 4.16. □

Recall that we want to enumerate all possible arc rotation permutations ρ and that the orbits of ρ are fixed. We have now reduced the problem of enumerating all maps to an enumeration of list permutations: By Lemma 4.2, each product of disjoint cycles *xss* corresponds to a permutation whose orbits are given by sset *xss*. So each product of disjoint cycles with the right sset yields a map. On the other hand, given the orbits *css* of f, Corollary 4.17 allows us to describe each permutation f by a product of disjoint cycles *xss* with sset *xss* = sset *css*.

The functions defined below compute all such list permutations. As rotations of a list induce the same permutation, we restrict ourselves to lists starting with the same element (this is justified by the map hd *xss* = map hd *css* result of Corollary 4.17):

Definition 4.18 (Permutations of Lists). *Let* product-lists *be the function computing for a list of lists all lists which can be obtained by choosing the n-th element from the n-th list.*

$$\begin{aligned}
\text{permutations } [] &:= [[]] \\
\text{permutations } xs &:= [y :: ys \mid y \in xs \wedge ys \in \text{permutations } (xs \setminus y)] \\
\\
\text{cyc-permutations } [] &:= [[]] \\
\text{cyc-permutations } (x :: xs) &:= \text{map } (\lambda ys.\ x :: ys)\ (\text{permutations } xs) \\
\\
\text{cyc-permutationss} &:= \text{product-lists} \circ \text{map cyc-permutations}
\end{aligned}$$

For a cycle *xs*, cycle (cyc-permutations *xs*) is the set of all permutations with orbit set *xs*. Similarly, for a product of disjoint cycles *xss* is cycles (cyc-permutationss *xss*) the set of all permutations with orbits sset *xss*.

Theorem 4.19. *Let xss be a list of disjoint cycles. Then*

$$\text{cycles } (\text{cyc-permutationss } xss) = \{f \mid f \text{ permutes } (\bigcup \text{sset } xss) \wedge (\forall xs \in xss.\ \text{cyclic-on } f\ xs)\}$$

Proof. Follows from Corollary 4.17 and the fact that cyc-permutationss computes all combinations of permutations of the lists in *xss*. □

We now have everything we need to enumerate the possible arc rotations ρ.

Lemma 4.20. *Let vs be a list of vertices and as a list of arcs such that* $G = \mathsf{LG}(vs, as)$ *is a graph. Then the set of all maps of G is*

$$\text{to-map } G\ (\text{cycles } (\text{all-maps } (vs, as)))$$

where

$$\text{grouped-out-arcs } (vs, as) := \text{map } (\lambda u.\ \text{filter } (\lambda(x, y).\ x = u)\ as)\ (\text{remdups } (\text{map fst } as))$$

is the list of arcs grouped by their tail and

$$\text{all-maps } (vs, as) := \text{cyc-permutationss } (\text{grouped-out-arcs } (vs, as)).$$

Proof. By Lemma 4.15, it suffices to show that

$$\text{cycles}\ (\text{all-maps}\ (vs, as)) = \{f \mid f\ \text{permutes}\ (\text{set}\ as) \wedge \text{rotates}_G\ f\}.$$

Let $xss = \text{grouped-out-arcs}\,(vs, as)$. Using Theorem 4.19, we still need to show that $\text{sset}\,xss = \mathsf{A}_G$ and $(\forall xs \in xss.\ \text{cyclic-on}\ f\ xs) \iff \text{rotates}_G\ f$. Both follow easily from the definition of grouped-out-arcs. □

This characterization is in the executable fragment of Isabelle/HOL, provided that *vs* and *as* are executable.

4.3.2. Computing the Genus

Let $M = (G, \pi, \rho)$ be a map. Recall the definition of the genus of a bidirected graph:

$$\text{genus}\ M = 2 \cdot |\text{sccs}\ G| - |\text{isolated-verts}\ G| - |\mathsf{V}_G| + |\mathsf{A}_G|/2 - |\mathsf{F}\ M|$$

To compute the genus, we need to be able to compute these five values. Computing the number of vertices, arcs and isolated vertices is trivial.

As G is a bidirected digraph, the reachability relation is symmetric and the SCCs can be computed by computing the reflexive-transitive closure of the reachability relation (cf. Lemma 3.18).

Definition 4.21 (Executable SCCs). *Let* $\text{scc}\ \ u = \{v \mid u \rightarrow_G^* v\}$ *and define*

$$\text{sccs}_{\text{impl}} \coloneqq \{\text{scc}\ \ u \mid u \in \mathsf{V}_G\}$$

Lemma 4.22. *Let G be a symmetric digraph, then* $\text{sccs}_{\text{impl}}\ G = \text{sccsV}\ G = \text{verts}\ (\text{sccs}\ G)$ *and hence* $|\text{sccs}\ G| = |\{\text{scc}\ \ u \mid u \in \mathsf{V}_G\}|$.

The expression $\{v \mid u \rightarrow_G^* v\}$ and hence the number of SCCs can easily be computed using the Executable Transitive Closures by Sternagel and Thiemann [72]. It remains now to give an executable specification for the number of face cycles. As before, I give a specification which uses lists to represent sets.

Definition 4.23 (Executable List of Orbits). *For a permutation f the* list orbit *is defined as*

$$\begin{aligned}\text{orbit}_{\text{impl}}\ f\ s\ acc\ x \coloneqq (&\text{let}\ x' = f\ x\\ &\text{in if}\ x' = s\ \text{then rev}\ (x :: acc)\ \text{else orbit}_{\text{impl}}\ f\ s\ (x :: acc)\ x')\end{aligned}$$

and the list of list orbits for a list of values as

$$\begin{aligned}\text{orbits}_{\text{impl}}\ f\ [] &\coloneqq []\\ \text{orbits}_{\text{impl}}\ f\ (x :: xs) &\coloneqq \text{let}\ fc = \text{orbit}_{\text{impl}}\ f\ x\ []\ x\ \text{in}\ fc :: \text{orbits}_{\text{impl}}\ f\ (xs \setminus fc).\end{aligned}$$

Note that $\text{orbit}_{\text{impl}}$ is only defined if s is in the orbit of x. Otherwise, it does not terminate. As it is a tail-recursive function, it can still be defined by the **partial_function** command of Isabelle/HOL. These functions can be used to compute $\mathsf{F}\ G$.

```
definition c_K5 ≔ ([0..4], [(x,y). x ← [0..4], y ← [0..4], x ≠ y])
lemma ¬comb_planar (𝒢 c_K5)
  by (subst comb_planar_impl_correct) eval+

definition c_K33 ≔ ([0..5], [(x,y). x ← [0..5], y ← [0..5], even x = odd y])
lemma ¬comb_planar (𝒢 c_K33)
  by (subst comb_planar_impl_correct) eval+
```

Figure 4.3.: Isabelle proof that two concrete instances of K_5 and $K_{3,3}$ are planar: Lemma 4.26 is applied and the resulting proof obligations can then be solved by simple evaluation. This includes the precondition that the graphs are symmetric and loop-free.

Lemma 4.24. *Let f be a permutation. Then* $\text{orbit}_{\text{impl}}$ *implements* orbit *and* $\text{orbits}_{\text{impl}}$ *implements* orbit *applied to a list of values. Formally, the equations* set $(\text{orbit}_{\text{impl}}\ f\ x\ []\ x) = \text{orbit}\ f\ x$ *and* sset $(\text{orbits}_{\text{impl}}\ f\ xs) = \text{orbit}\ (\text{set}\ xs)$ *hold. Moreover, if xs is distinct,* $\text{orbit}_{\text{impl}}\ f\ xs$ *is also distinct.*

Proof. By induction, using the equation orbit $f\ x = \{(^n\ x.\ n \in \mathbb{N}\}$. □

With these functions, I can now give an executable specification for combinatorial planarity.

Definition 4.25 (Executable Combinatorial Planarity). *I implement* comb-planar *by*

$$\begin{aligned}\text{comb-planar}_{\text{impl}}\ (vs, as) \coloneqq\ & \text{let } G = \text{LG}(vs, as)\\ & \text{let } n = 2 \cdot |\text{sccs}_{\text{impl}}\ (vs, as)| - |\text{isolated-verts } G| - |\text{V}_G| + |\text{A}_G|/2\\ & \text{in } \exists xss \in \text{all-maps } G.\ n - |\text{orbits}_{\text{impl}}\ (\sigma_{\text{to-map } G\ xss})\ as| = 0.\end{aligned}$$

Lemma 4.26. *Let* $\text{LG}(vs, as)$ *be a symmetric and loop-free digraph. Then*

$$\text{comb-planar}\ (\text{LG}(vs, as)) = \text{comb-planar}_{\text{impl}}\ (vs, as)$$

holds.

Proof. By Lemma 4.24, $|\text{F}\ (\text{to-map}\ G\ xss)| = |\text{orbits}_{\text{impl}}\ (\sigma_{\text{to-map } G\ xss})\ as|$. The lemma then follows by unfolding the lets, Lemma 4.22, and Lemma 4.20. □

With this executable specification, combinatorial planarity for a concrete K_5 and $K_{3,3}$ can be proven by evaluation. Figure 4.3 shows the Isabelle proof text.

4.4. Kuratowski Graphs are not Combinatorially Planar

In the previous section, we saw for concrete instances of the $K_{3,3}$ and K_5 that these graphs are not combinatorially planar. We still need to generalize this result to arbitrary instances of $K_{3,3}$

and K_5. The proof idea is simple: All $K_{3,3}$ respectively K_5 are isomorphic and the property of being combinatorially planar is a graph invariant.

If G and H are digraphs without parallel arcs, any isomorphism between G and H is uniquely determined by the vertex morphism. To determine the arc morphism, one first maps the endpoints with the vertex morphism and then obtains the unique arc with these endpoints. This construction is performed by the function defined below.

Definition 4.27. *Let G, H be digraphs without parallel arcs and $f : \mathsf{V}_G \to \mathsf{V}_H$ injective. Then the* isomorphism induced by f *is defined as*

$$\mathsf{iso}_{\mathsf{nomulti}}\ G\ H\ f \coloneqq (f, (\mathsf{ends}\ H)^{-1} \circ \mathsf{pairself}\ f \circ \mathsf{ends}\ G, \mathsf{tail}\ H, \mathsf{head}\ H)$$

where for any graph G, $\mathsf{ends}\ G : \mathsf{A}_G \to \mathsf{V}_G \times \mathsf{V}_G$ *is defined by* $\mathsf{ends}\ G\ a = (a_{\mathsf{t},G}, a_{\mathsf{h},G})$ *and* pairself *lifts a function to a pair by* $\mathsf{pairself}\ f\ (u, v) = (f\ u, f\ v)$.

Lemma 4.28. *Let G, H be digraphs without parallel arcs. If $f : \mathsf{A}_G \to \mathsf{A}_G$ is injective and $\mathsf{adj}_H = \mathsf{pairself}\ f\ \mathsf{adj}_G$, then $\mathsf{iso}_{\mathsf{nomulti}}\ G\ H\ f$ is an isomorphism between G and H.*

Using $\mathsf{iso}_{\mathsf{nomulti}}$, it is easy to show that arbitrary instances of K_m and $\mathsf{K}_{m,n}$ are isomorphic:

Lemma 4.29. *Let G, H be graphs with $\mathsf{K}_m\ G$ and $\mathsf{K}_m\ H$. Then G and H are isomorphic.*

Proof. Both G and H have the same, finite cardinality, so we obtain a bijection $f : \mathsf{V}_G \to \mathsf{V}_H$. As both G and H are complete, the image of adj_G under $\mathsf{pairself}\ f$ is adj_H. By Lemma 4.28, G and H are isomorphic. □

Lemma 4.30. *Let G, H be graphs with $\mathsf{K}_{m,n}\ G$ and $\mathsf{K}_{m,n}\ H$. Then G and H are isomorphic.*

Proof. We choose functions f_1 resp. f_2 for the partitions with cardinality m resp. n separately. Combining these yields a vertex morphism obeying the bipartition and hence the adjacency relation. □

It is left to show that combinatorial planarity is a graph invariant. Recall that the genus is computed from the following properties: the number of vertices, the number of arcs, the number of face cycles, the number of SCCs and the number of isolated vertices. Except for face cycles, all of these properties are graph invariants according to Section 3.3.3.

The number of face cycles is defined with regard to a combinatorial map, not just a graph. A digraph isomorphism can be extended to maps.

Definition 4.31. *Let $M = (G, \pi_G, \rho_H)$ be map, h a digraph isomorphism for G, and define $h^{-1} \coloneqq \mathsf{inv\text{-}iso}_G\ h$. A function $f : \mathsf{A}_G \to \mathsf{A}_G$ can be lifted to a function $\mathsf{A}_{h\,G} \to \mathsf{A}_{h\,G}$ by the following function:*

$$\mathsf{wrap}\ h\ f = \mathsf{perm\text{-}restrict}\ (h \circ f \circ h^{-1})\ \mathsf{A}_{h\,G}$$

The application of h to M is then defined as

$$\mathsf{map\text{-}iso}\ h\ M \coloneqq (h\ G, \mathsf{wrap}\ h\ \pi_H, \mathsf{wrap}\ h\ \rho_H).$$

Lemma 4.32. *Let $M = (G, \pi_G, \rho_H)$ be a map and h an graph isomorphism for G. Then $M' =$ map-iso h M is a map with $\sigma_{M'} =$ wrap h σ_M.*

Proof. Easy. The equation $\sigma_{M'} =$ wrap $h\sigma_M$ follows directly from the definition of wrap and the injectivity of iso-arcs h. □

Corollary 4.33. *Combinatorial planarity is a graph invariant.*

Proof. Let G, H be isomorphic digraphs and $M_G = (G, \pi_G, \rho_G)$ a map. Then there is an isomorphism h between G and H, i.e., $H = h\,G$. Let $M_H =$ map-iso h $M = (H, \pi_H, \rho_H)$. Then for all $a \in \mathsf{A}_H$,

$$\text{orbit } \sigma_{M_H}\ a = \text{iso-arcs } h\ (\text{orbit } \sigma_M\ (\text{iso-arcs } (\text{inv-iso}_G\ h)\ a))$$

and hence

$$\mathsf{F}\ M_H = \text{iso-arcs } h\ (\mathsf{F}\ M_G).$$

As h is an isomorphism, iso-arcs h is injective on $\mathsf{F}\ M_G$ and thus $|\mathsf{F}\ M_G| = |\mathsf{F}\ M_H|$ holds. As noted above, all other components of the genus of a map are graph invariants, hence genus $M_G =$ genus M_H.

So plane maps of G are mapped to plane maps of H and therefore, G is combinatorially planar if and only if H is. □

Corollary 4.34. *Kuratowski graphs are not combinatorially planar.*

Proof. Follows directly by applying Corollary 4.33, Lemma 4.29, and Lemma 4.30 to the graphs from Figure 4.3. □

4.5. Planarity under Subdivision

In this section, I show that the subdivision of a Kuratowski graph is not combinatorially planar. Generalizing this statement, I prove the following proposition:

Proposition 4.35. *Let H be a subdivision of G and G combinatorially planar. Then H is combinatorially planar.*

That is, a subdivision of a non-planar graph is non-planar. The idea of the proof is that a subdivision step preserves planarity and hence, by induction, subdivision does. For the intuitive definition of planarity, the induction step seems obvious. Still, the formal proof that combinatorial planarity is preserved by subdivision is far from being trivial.

A subdivision step is described by the predicate subdiv-step which establishes a complex relationship between two graphs – unfolding this predicate is not a feasible approach. We are only ever interested in a single instance of the subdivision step, so we introduce a locale to collect all the facts about a map derived by a subdivision step.

locale subdiv_map =
fixes $G\ \pi_G\ H\ \pi_H\ \rho_H\ u\ v\ w\ uv\ uw\ vw$
assumes subdiv_step (G, π_G) (H, π_H) (u, v, w) (uv, uw, vw)
assumes digraph_map (H, π_H, ρ_H)

For the rest of this chapter, we will work in this locale and write $vu = \pi_G\ uv$, $wu = \pi_H\ uw$, and $wv = \pi_H\ vw$.

Using a locale allows us to derive unconditional facts and equations. This makes automatic reasoning tools much more effective. In particular, such facts can guide these tools to perform necessary case analysis automatically. For example, given the fact that $\{uw, wu, vw, wv\} \subseteq \mathsf{A}_H$ and the premise that $x \in \mathsf{A}_H$, many of Isabelle's automated reasoning tools will analyze the cases $x = uw$, $x = wu$, $x = vw$, $x = wv$, and $x \notin \{uw, wu, vw, wv\}$.

In order to prove Proposition 4.35, I construct a plane map for G from a plane map for H. As w was inserted into H by a subdivision, there is necessarily a face cycle involving the sequence uw, wv and one involving the sequence vw, wu:

Lemma 4.36. *For H holds $\sigma_H\ uw = wv$ and $\sigma_H\ vw = wu$.*

Proof. As w has only two outgoing arcs in H, we have $\rho_H\ wu = wv$ and $\rho_H\ wv = wu$. The lemma follows then by unfolding the definition of σ_H. □

By replacing the sequence uw, wv by uv and vw, wu by vu in σ_H, we get a map for G.

Lemma 4.37. *Let*

$$\rho_G \coloneqq \text{perm-swap } uw\ uv\ (\text{perm-swap } vw\ vu\ (\text{perm-rem } wv\ (\text{perm-rem } wu\ \rho_H))).$$

Then (G, π_G, ρ_G) is a map for G and σ_G is defined as $\sigma_G \coloneqq \rho_G \circ \pi_G$.

The arcs wu and wv are removed as there is no vertex w in G anymore (those two arcs are the only outgoing arcs of w in H). Then, the arc uw is replaced by uv and vw by vu.

The face cycles of G can be easily expressed in terms of the face cycles of H.

Definition 4.38 (Projection of Arcs). *We define projection functions $\text{proj}_H : \mathsf{A}_H \to \mathsf{A}_G$ and $\text{proj}_G : \mathsf{A}_G \to \mathsf{A}_H$ as follows:*

$$\text{proj}_H\ x \coloneqq \begin{cases} uv & \text{if } x \in \{uw, wv\} \\ vu & \text{if } x \in \{vw, wu\} \\ x & \text{else} \end{cases} \qquad \text{proj}_G\ x \coloneqq \begin{cases} uw & \text{if } x = uv \\ vw & \text{if } x = vu \\ x & \text{else} \end{cases}$$

Lemma 4.39. *The function proj_H relates the face cycles of G and H: $\mathsf{F}_G = \text{proj}_H\ \mathsf{F}_H$.*

Proof. The following three equations hold (for $a \in \mathsf{A}_G \setminus \{uv, vu\}$):

$$\sigma_G\ a = \text{proj}_H\ (\sigma_H\ a) \qquad \sigma_G\ uv = \text{proj}_H\ (\sigma_H\ wv) \qquad \sigma_G\ vu = \text{proj}_H\ (\sigma_H\ wu)$$

With Lemma 4.36, one can now easily show that

$$\begin{aligned} \text{orbit } \sigma_G\ a &= \text{proj}_H\ (\text{orbit } \sigma_H\ (\text{proj}_G\ a)) && \text{for all } a \in \mathsf{A}_G \\ \text{proj}_H\ (\text{orbit } \sigma_H\ a) &= \text{orbit } \sigma_G\ (\text{proj}_H\ a) && \text{for all } a \in \mathsf{A}_H \end{aligned}$$

and the lemma follows directly from these two properties. □

Lemma 4.40. *G and H have the same number of face cycles, i.e., $|\mathsf{F}_G| = \mathsf{F}_H|$.*

Proof. By Lemma 4.39 it suffices to show that proj_H is injective on F_H. Let $A, B \subseteq \mathsf{F}_H$ and $\text{proj}_H\, A = \text{proj}_H\, B$. Then $\text{proj}_H^{-1}\,(\text{proj}_H\, X) - \{uv, vu\} = X$ holds for $X \in A, B$: the inverse image of uv under proj_H is $\{uv, uw, wv\}$ and $uv \notin X$ (as $uv \notin \mathsf{A}_H$) and either both or neither of uw and wv are in X (as $\sigma_H\ uw = wv$). The same holds for vu and $\{vu, vw, wu\}$.

Hence $A = B$ and f is injective. □

So (G, π_G, ρ_G) is an appropriate choice for the map for G: the number of face cycles is the same as for H, which is what one would expect for inserting a vertex on an arc.

To compute the genus we need to know the number of vertices, arcs, isolated vertices, face cycles, and (strongly connected) components. The first three values are easy to compute, and we have just shown that the fourth stays the same, so we now consider the SCCs of G. Intuitively, those are the same as the SCCs of H, except for the removed vertex w (which, by construction of the subdivision is in the same SCC as u and v).

To relate $\text{sccsV}\ G$ and $\text{sccsV}\ H$, we need to consider the relationship between $\rightarrow_G^*$ and $\rightarrow_H^*$. It is relatively easy to see that these relations are same, except when w is concerned. As above, this can be hidden by a projection function.

Lemma 4.41. *Let $\text{projV}_H : \mathsf{A}_G \to \mathsf{A}_H$ be defined as*

$$\text{projV}_H\ x = \begin{cases} u & \text{if } x = w \\ x & \text{else.} \end{cases}$$

Then, for all x, y holds $x \rightarrow_H^* y$ if and only if $\text{projV}_H\ x \rightarrow_G^* \text{projV}_H\ y$ holds.

Proof. Both directions can be proven by an easy induction on the definition of reachability. □

Using this characterization, it is easy to to show that the number of components in G and H is the same:

Lemma 4.42. *G and H have the same number of components, i.e., $|\text{sccsV}\ G| = |\text{sccsV}\ H|$.*

Proof. Recall that the SCCs of a graph are fully characterized by the reachability relation (cf. Lemma 3.18). Hence, from Lemma 4.41 the equation $\text{sccsV}\, G = \text{projV}_H\,(\text{sccsV}\, H)$ follows directly.

As the SCCs of a graph are disjoint and w is always in the same SCC as u, projV_H is injective on the SCCs of H. Hence, $|\text{projV}_H\ (\text{sccsV}\ H)| = |\text{sccsV}\ H|$ and the lemma is proven. □

The function projV_H allows a concise proof of Lemma 4.42: It hides any considerations about w, except for the injectivity proof.

Corollary 4.43. *The maps (G, π_G, ρ_G) and (H, π_H, ρ_H) have the same genus.*

Proof. G has one vertex and two arcs less then G. The number of isolated vertices, face cycles and SCCs stays the same. Therefore this lemma holds. □

From this we can derive our final theorem, without the preconditions of the subdiv_step locale:

Theorem 4.44 (Planarity of Subdivisions). *Let (H, π_H) be a subdivision of (G, π_H). If H is combinatorially planar, then G is also combinatorially planar.*

Proof. As H is combinatorially planar, we obtain a plane map (H, π'_H, ρ_H). Then there is also a π'_G such that subdivision (G, π'_G) (H, π'_H). By induction over this predicate and Corollary 4.43, we obtain a plane map for G. Hence, G is combinatorially planar. □

4.6. Planarity under Subgraphs

In this section, I show that every bidirectable subgraph H (i.e., a subgraph for which there is a bidirection) of a combinatorially planar graph G is combinatorially planar. If H is a bidirectable subgraph of G, it can be constructed by repeatedly applying one of the following steps:

- removing an isolated vertex or
- removing an edge (i.e., an arc and its reversal w.r.t. some bidirection).

If we can show that both steps preserve planarity, we know that H must be planar. It is easy to show that removing an isolated vertex preserves planarity:

Lemma 4.45. *Let G be a digraph with a map M and let u be an isolated vertex of G. Then, M is also a map of the digraph $H = G - \{u\}$ and the genus of H is the same as the genus of G.*

Proof. As u is an isolated vertex, G and H have the same arcs and hence M is also a map of H. Therefore, G and H have the same number of arcs and face cycles. H has one vertex, one isolated vertex and one SCC less then G, hence G and H have the same genus. □

For the remainder of this section let G be an arbitrary, but fixed combinatorially planar graph with map (π_G, ρ_G) and $a \in \mathsf{A}_G$. We now consider the graph $G - \{a, \pi_G\, a\}$ and fix some notation.

Definition 4.46. *We write $a' = \pi_G\, a$ and $H \coloneqq G - \{a, a'\}$. Moreover, we define*

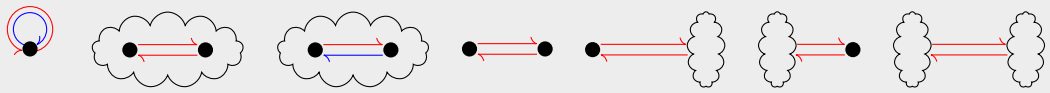
$$\pi_H = \text{perm-restrict}\ \pi_G\ (\mathsf{A}_G - \{a, a'\})$$
$$\rho_H = \text{perm-rem}\ a\ (\text{perm-rem}\ a'\ \rho_G).$$

Lemma 4.47. *(H, π_H, ρ_H) is a map.*

Proof. Easy. □

To compute the genus of H, we will have to do a case analysis on four properties:

- *B:* a and a' may be a bridge in their SCC (i.e., removing these splits the component into two). By abuse of notation, we call G *biconnected* if a is reachable from a' and vice versa in H.
- *T:* $a_{\mathsf{t},H}$ may be isolated in H.
- *H:* $a_{\mathsf{h},H}$ may be isolated in H.



(a) BT (b) BtS (c) Bts (d) bTH (e) bTh (f) btH (g) bth

Figure 4.4.: All possible situations for removing an edge from a bidirected graph. A ● represents a single vertex, a ☁ a SCC, and the arrows represent the arcs to be deleted. The color indicates whether the arrows lay on the same face cycle. The letters indicate which of the four properties are assumed to hold (uppercase) or not to hold (lowercase).

- *S:* a and a' may be on the same face cycle.

Not all combinations of the four points above are possible, so we end up with the seven different situations depicted in Figure 4.4.

In Isabelle, I use locales to structure the proof: A basic locale contains all necessary properties about G and H which are independent from the four properties above. Then there are two locales for each of the four properties, inheriting from the basic locale. Finally, there is one locale for each of the seven possible situations, inheriting from the locales for the properties (sometimes using intermediate locales). Each of these seven locales ends with a lemma stating genus H $(\pi_H, \rho_H) \leq$ genus G (π_G, ρ_G). The final theorem then follows from the case analysis depicted in Figure 4.4.

Lemma 4.48. *Let G be biconnected. Then* $|\text{sccs } H| = |\text{sccs } G|$. *If G is not biconnected, then* $|\text{sccs } H| = |\text{sccs } G| + 1$.

Proof. If G is biconnected, then u is reachable from v in G if and only if it is reachable in H and hence the SCCs of G and H are the same.

Let us now assume that G is not biconnected. We write $\text{scc}_G\ u$ for the vertex SCC of G containing u. Then obviously $a_{\text{t},G} \neq a_{\text{h},G}$ and $\text{scc}_H\ a_{\text{t},G} \neq \text{scc}_H\ a_{\text{h},G}$. Moreover, $\text{scc}_G\ a_{\text{t},G} = \text{scc}_H\ a_{\text{t},G} \cup \text{scc}_H\ a_{\text{h},G}$ and for all vertices u outside of this SCC, we have have $\text{scc}_G\ u = \text{scc}_H\ u$. Hence,

$$\text{sccs } H = (\text{sccs } G \setminus \{\text{scc}_G\ a_{\text{t},G}\}) \cup \{\text{scc}_H\ a_{\text{t},G}, \text{scc}_H\ a_{\text{h},G}\}$$

and therefore $|\text{sccs } H| = |\text{sccs } G| + 1$. □

We now consider the face cycles of H. It can be easily shown that $\sigma_G\ x = \sigma_H\ x$ except when $x \in \{a, a'\}$ or $\sigma_G\ x \in \{a, a'\}$. Hence, the face cycles of G can be characterized as follows:

Lemma 4.49. *All face cycles of G which contain neither a nor a' are also face cycles of H. To be exact, the following equation characterizes the face cycles of H:*

$$\mathsf{F}\ H = \mathsf{F}\ G \setminus \{\text{orbit } \sigma_G\ a, \text{orbit } \sigma_G\ a'\} \cup \{\text{orbit } \sigma_H\ ((\text{orbit } \sigma_G\ a \cup \text{orbit } \sigma_G\ a') \setminus \{a, a'\})\}$$

Proof. As $\sigma_G\ x = \sigma_H\ x$ except when $x \in \{a, a'\}$ or $\sigma_G\ x \in \{a, a'\}$, the face cycles of G containing neither a nor a' are also face cycles of H. Hence, the left side of the union is a subset of $\mathsf{F}\ H$. As $\mathsf{A}_G = \mathsf{A}_H \setminus \{a, a'\}$, the right side of the union is trivially a subset of $\mathsf{F}\ H$.

Any element of $\mathsf{F}\ H$ is also contained in the right hand side, as there is an H-face cycle for each arc of H. □

We will now have a closer look at the face cycles of a and a'. Even for those face cycles, the parts "between a and a'" are not affected by deleting these edges. This notion of "between two vertices" is expressed by segments. If S is the face cycle containing a and a', it can be split into four parts:

$$S = \text{segment}\ \sigma_G\ a\ a' \cup \{a'\} \cup \text{segment}\ \sigma_G\ a'\ a \cup \{a\}$$

As a, a' are not arcs of H, the remaining face cycle(s) of H must of the other two parts. The drawings in Figure 4.4 suggest that each of these segments is already a face cycle of H. This is shown in the following lemma.

Lemma 4.50. *Let* orbit $\sigma_G\ a = \text{orbit}\ \sigma_G\ a'$. *If* segment $\sigma_G\ a'\ a \neq \varnothing$, *then it is a face cycle of* H. *The same holds for* segment $\sigma_G\ a\ a'$.

Proof. Consider segment $\sigma_G\ a'\ a$. As the segment is not empty, we have $\sigma_G\ a' \notin \{a, a'\}$ and hence $\sigma_G\ a' \in \mathsf{A}_H$.

Moreover, for all elements x of this segment (except for the "last", i.e., the one with $\sigma_G\ x = a$) holds $\sigma_H\ x = \sigma_G\ x$.

We show now that in H the last element of the segment loops around to the first, i.e., that $\sigma_G\ x = a$ implies $\sigma_H\ x = \sigma_G\ a'$. From $x \notin \{a, a'\}$ follows $\pi_H(x) = \pi_G(x)$. Then holds:

$$\begin{aligned}
\sigma_G\ a' &= \sigma_G\ (\pi_G\ a) = \rho_G\ a && \text{as } \sigma_G = \rho_G \circ \pi_G \\
&= \rho_G\ (\sigma_G\ x) \\
&= \rho_G\ (\rho_G\ (\pi_G\ x)) \\
&= \rho_H\ (\pi_G\ x) && \text{from } \rho_G\ (\pi_G\ x) = a \text{ and def. of perm-rem} \\
&= \rho_H\ (\pi_H\ x) = \sigma_H\ x
\end{aligned}$$

Then both directions of the equation segment $\sigma_G\ a'\ a = \text{orbit}\ \sigma_H\ (\sigma_G\ a')$ can be easily proven by induction on the definition of segments respectively orbits. As $\sigma_G\ a' \in \mathsf{A}_H$, the segment is a face cycle of H.

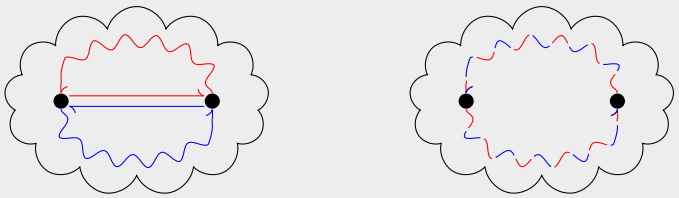
For segment $\sigma_G\ a\ a'$, the same holds due to symmetry. □

Hence, if a and a' lie on the same face cycle, we now know the number of face cycles and it depends only on whether these segments are empty or not:

Lemma 4.51. *Let* orbit $\sigma_G\ a = \text{orbit}\ \sigma_G\ a'$. *Then*

$$|\mathsf{F}\ H| = |\mathsf{F}\ G| - 1 + |\{\text{segment}\ \sigma_G\ a'\ a, \text{segment}\ \sigma_G\ a'\ a\} - \{\varnothing\}|$$

Proof. Follows from Lemmas 4.49 and 4.50. □



(a) in G

(b) in H, after arcs a and a' were removed

Figure 4.5.: Situation Bts: a and a' are on different, non-trivial face cycles in G. These are merged in H.

Now consider the case where a and a' are on different face cycles. In this case, removing a and a' combines their two face cycles into a single one (see Figure 4.5).

Lemma 4.52. *Let* orbit $\sigma_G\ a \neq$ orbit $\sigma_G\ a'$. *Let* $b \in A_H$ *such that* $b \in$ orbit $\sigma_G\ a \cup$ orbit $\sigma_G\ a'$. *Then*

$$\text{orbit}\ \sigma_H\ b = \text{segment}\ \sigma_G\ a\ a \cup \text{segment}\ \sigma_G\ a'\ a'$$

Proof. In this proof we assume that $b \in$ orbit $\sigma_G\ a$. The proof for $b \in$ orbit $\sigma\ G\ a'$ is analogous.

Case $\subseteq$*:* Let $x \in$ orbit $\sigma_H\ b$. Assume that $x \notin$ orbit $\sigma_G\ a \cup$ orbit $\sigma_G\ a'$. Then holds

$$\text{orbit}\ \sigma_G\ x \cap (\text{orbit}\ \sigma_G\ a \cup \text{orbit}\ \sigma_G\ a') = \varnothing$$

and hence by Lemma 4.49 orbit $\sigma_G\ x =$ orbit $\sigma_H\ x =$ orbit $\sigma_H\ b$.

As a result, $b \notin$ orbit $\sigma_G\ a \cup$ orbit $\sigma_G\ a'$, which is a contradiction to the assumption of the Lemma. Hence $x \in$ orbit $\sigma_G\ a \cup$ orbit $\sigma_G\ a'$. For arbitrary f and x holds segment $f\ x\ x =$ orbit $f\ x \setminus \{x\}$ and with $x \notin \{a, a'\}$ follows $x \in$ segment $\sigma_G\ a\ a \cup$ segment $\sigma_G\ a'\ a'$.

Case $\supseteq$*:* As $b \in$ orbit $\sigma_G\ a$, we have orbit $\sigma_G\ a =$ segment $\sigma_G\ a\ b \cup \{b\} \cup$ segment $\sigma_G\ b\ a$. This case is split in four parts:

- $b \in$ orbit $\sigma_H\ b$: Trivial.
- segment $\sigma_G\ b\ a \subseteq$ orbit $\sigma_H\ b$: We have $a \notin$ segment $\sigma_G\ b\ a$ and, as orbit $\sigma_G\ a \neq$ orbit $\sigma_G\ a'$, $a' \notin$ segment $\sigma_G\ b\ a$. Hence, $\sigma_G\ x = \sigma_H\ x$ for all $x \in$ orbit σ_G until $\sigma_G\ x = a$ and the case follows by induction on the definition of segments.
- segment $\sigma_G\ a'\ a' \subseteq$ orbit $\sigma_H\ b$: Let $c = \sigma_G^{-1}\ a$. Then $c \in \{b\} \cup$ segment $\sigma_G\ b\ a$ and, by the previous cases, $c \in$ orbit $\sigma_H\ b$. Now $\sigma_G\ a' = \sigma_H\ c$ can be shown. Hence, $\sigma_G\ x = \sigma_H\ x$ for all $x \in$ orbit $\sigma_G\ a'$ until $\sigma_G\ x = a'$ and by induction on the definition of segments we have segment $\sigma_G\ a'\ a' \subseteq$ orbit $\sigma_H\ c$. With $c \in$ orbit $\sigma_H\ b$ follows orbit $\sigma_H\ c =$ orbit $\sigma_H\ b$ and thus the case.
- segment $\sigma_G\ a\ b \subseteq$ orbit $\sigma_H\ b$: Let $x \in$ segment $\sigma_G\ a\ b$.

There is a d such that $d \in \text{orbit } \sigma_H\ b$ and $\sigma_H\ d = \sigma_G\ a$. If segment $\sigma_G\ a'\ a' = \varnothing$, $d = c$. Otherwise, $d = \sigma_G^{-1}\ a'$.

As $x \in \text{segment } a\ b$ we have $x \neq a$ and with $b \neq a$ follows $\sigma_G\ x \neq a$. Moreover segment $\sigma_G\ a\ b \cap \text{orbit } \sigma_G\ a' = \varnothing$ and therefore $\{x, \sigma_G\ x\} \cap \{a, a'\} = \varnothing$. As a result, $\sigma_G\ x = \sigma_H\ x$ for all elements of segment $\sigma_G\ a\ b$ up to and including x. Moreover, we have $\sigma_H\ d = \sigma_G\ a$, so $x \in \text{orbit } \sigma_H\ d$ follows by induction.

We know that $d \in \text{orbit } \sigma_H\ b$ and hence by transitivity $x \in \text{orbit } \sigma_H\ b$. This proves the case.

□

As a result, the number of face cycles is then given by the following lemma.

Lemma 4.53. *Let* $\text{orbit } \sigma_G\ a \neq \text{orbit } \sigma_G\ a'$. *Then*

$$|\mathsf{F}\ H| = |\mathsf{F}\ G| - \begin{cases} 1 & \text{if segment } \sigma_G\ a\ a \cup \text{segment } \sigma_G\ a'\ a' \neq \varnothing \\ 2 & \text{else} \end{cases}$$

Proof. By Lemma 4.49 and Lemma 4.52. □

We are now able to show that removing a pair of arcs decreases the genus:

Lemma 4.54. *The genus of* H *is bounded by the genus of* G*:*

$$\text{genus } (H, \pi_H, \rho_H) \leq \text{genus } (G, \pi_G, \rho_G)$$

Proof. We write

$$\begin{aligned} \delta_{\text{sccs}} &:= |\text{sccs } G| - |\text{sccs } H| \\ \delta_{\text{isolated-verts}} &:= |\text{isolated-verts } G| - |\text{isolated-verts } H| \\ \delta_{\mathsf{F}} &:= |\mathsf{F}\ G| - |\mathsf{F}\ H| \\ \delta_{\mathsf{A}} &:= |\mathsf{A}_G| - |\mathsf{A}_H| = 2 \\ \delta_{\text{genus}} &= 2 \cdot \delta_{\text{sccs}} - \delta_{\text{isolated-verts}} - \delta_{\mathsf{F}} + \delta_{\mathsf{A}}/2 \end{aligned}$$

By the definition of the genus holds $\text{genus } (H, \pi_H, \rho_H) \leq \text{genus } (G, \pi_G, \rho_G)$ if $0 \leq \delta_{\text{genus}}$.

If G is not biconnected, then a and a' are on the same face cycle: both a and the arcs in segment $\sigma_G\ a'\ a'$ describe an non-empty path from $a_{\mathsf{t},G}$ to $a_{\mathsf{h},G}$ in G. However, $a_{\mathsf{t},G}$ and $a_{\mathsf{h},G}$ are not connected in H, hence $a \in \text{segment } \sigma_G\ a'\ a'$ and therefore $\text{orbit } \sigma_G\ a = \text{orbit } \sigma_G\ a'$.

We distinguish the seven cases depicted in Figure 4.4. For any of the cases, the number of SCCs follows directly from Lemma 4.48.

- *G is biconnected and $a_{\mathsf{t},G}$ is isolated in H (Figure 4.4a):* As $a_{\mathsf{t},G}$ is isolated and connected to $a_{\mathsf{h},G}$, these two vertices are equal, hence $\delta_{\text{isolated-verts}} = -1$. The face cycles of a and a' are $\{a\}$ and $\{a'\}$, therefore

  $$\text{segment } \sigma_G\ a\ a = \text{segment } \sigma_G\ a'\ a' = \varnothing$$

  and by Lemma 4.53 we have $\delta_{\mathsf{F}} = 2$. Moreover, $\delta_{\text{sccs}} = 0$ and hence $\delta_{\text{genus}} = 0$.

- *G is biconnected, $a_{t,G}$ is not isolated in H and a and a' are on the same face cycle (Figure 4.4b):* Then $a_{h,G}$ is not isolated either (because $a_{h,G}$ and $a_{t,G}$ are connected), so $\delta_{\text{isolated-verts}} = 0$. From Lemma 4.51 follows directly $\delta_F \leq 1$. As $\delta_{\text{sccs}} = 0$, we have $0 \leq \delta_{\text{genus}}$.

- *G is biconnected, $a_{t,G}$ is not isolated in H and a and a' are not on the same face cycle (Figure 4.4c):* As above, $\delta_{\text{isolated-verts}} = 0$ and $\delta_{\text{sccs}} = 0$.

  We show that segment $\sigma_G\ a\ a \cup$ segment $\sigma_G\ a'\ a' \neq \varnothing$. Then, $\delta_F = 1$ holds by Lemma 4.53 and hence $0 = \delta_{\text{genus}}$ follows.

  We assume that the two segments are empty. Then we have $\sigma_G\ a = a$ and $\sigma_G\ a' = a'$ and hence $\rho_G\ a = a'$ and $\rho_G\ a' = a$. As ρ permutes the arcs around a vertex, it follows that a, a' are the only outgoing arcs of $a_{t,G}$ in G. Hence, in H, $a_{t,G}$ must be isolated. This is a contradiction.

- *G is not biconnected and $a_{t,G}$ and $a'_{h,G}$ are isolated in H(Figure 4.4d):* Then $\delta_{\text{isolated-verts}} = -2$, $\delta_{\text{sccs}} = -1$ and $\delta_F = 1$ by Lemma 4.51. Hence $\delta_{\text{genus}} = 0$.

- *G is not biconnected, $a_{t,G}$ is isolated in H and $a'_{h,G}$ is not isolated in H (Figure 4.4e):* Then $\delta_{\text{isolated-verts}} = -1$, $\delta_{\text{sccs}} = -1$ and by Lemma 4.51 $\delta_F = 0$. Hence $\delta_{\text{genus}} = 0$.

- *G is not biconnected, $a_{t,G}$ is not isolated in H and $a'_{h,G}$ is isolated in H (Figure 4.4f):* From the previous case by symmetry.

- *G is not biconnected and neither $a_{t,G}$ nor $a'_{h,G}$ are isolated in H (Figure 4.4g):* In this case, $\delta_{\text{isolated-verts}} = 0$, $\delta_{\text{sccs}} = -1$ and $\delta_F = -1$ by Lemma 4.51. Hence $\delta_{\text{genus}} = 0$.

□

Now, we can use induction to show that every bidirectable subgraph H of a combinatorially planar graph G is combinatorially planar. Note: As H is bidirectable and G is combinatorially planar, there are arc reversal functions π_H and π_G for H and G respectively. In general, these are not compatible in the sense that $\pi_H\ a = \pi_G\ a$ for all $a \in A_H$. Lemma 4.54, however, is only applicable if these are compatible. To solve this, one could prove that if G is combinatorially planar there is a plane map for every arc reversal π'_G function of G and then choose π'_G such that it is compatible of π_H. This would involve induction over some notion of distance between π_G and π'_G.

Instead, I embed the modification of the arc reversal function in the inductive step of the planarity proof; modifying π_G only one arc at a time.

Lemma 4.55. *Let G be a graph with map (π_G, ρ_G). Let $\{a, b\} \subseteq A_G$ with $a_{t,G} = b_{t,G}$ and $a_{h,G} = b_{h,G}$. Let $\pi'_G =$ perm-swap $\pi_G\ a\ b$ and $\rho'_G =$ perm-swap $\rho_G\ a\ b$. Then (π'_G, ρ'_G) is a map for G with* genus $(G, \pi'_G, \rho'_G) =$ genus (G, π_G, ρ_G)

Lemma 4.56. *Let (G, π_G, ρ_G) be a map and H be a bidirectable subgraph of G. Then there is a map (π_H, ρ_H) of H such that* genus $H\ (\pi_H, \rho_H) \leq$ genus $G\ (\pi_G, \rho_G)$.

Proof. Let $\delta_G = |\mathsf{A}_G| - |\mathsf{A}_H| + |\mathsf{V}_G| - |\mathsf{V}_H|$. Then show that for all G the rule

$$\frac{\text{subgraph } H\ G \quad (G, \pi_G, \rho_G) \text{ is map}}{\exists(\pi_H, \rho_H).\ (H, \pi_H, \rho_H) \text{ is map} \wedge \text{genus } H\ (\pi_H, \rho_H) \leq \text{genus } G\ (\pi_G, \rho_G)}$$

holds by induction on δ_G.

Case $\delta_G = 0$*:* Then $G = H$ and the property holds trivially.

Case $|\mathsf{A}_G| > |\mathsf{A}_H|$*:* Then $\mathsf{A}_H \subsetneq \mathsf{A}_G$ and there are $\{a, a'\} \subseteq \mathsf{A}_H \setminus \mathsf{A}_G$, such that $a \neq a'$, $a'_{\mathrm{t},G} = a_{\mathrm{h},G}$, and $a'_{\mathrm{h},G} = a_{\mathrm{t},G}$ (as both G and H are bidirectable). By Lemma 4.55,

$$\pi'_G \coloneqq \text{perm-swap } \pi_G\ a'\ (\pi_G\ a) \qquad \rho'_G \coloneqq \text{perm-swap } \rho_G\ a'\ (\pi_G\ a)$$

are a map for G with genus $(G, \pi'_G, \rho'_G) = \text{genus } (G, \pi_G, \rho_G)$. Then, by Lemma 4.54, there is a map such that

$$\text{genus } (G - \{a, a'\}, \pi_{G-\{a,a'\}}, \rho_{G-\{a,a'\}}) \leq \text{genus } (G, \pi_G, \rho_G)$$

Moreover, H is a subgraph of $G - \{a, a'\}$ and hence from the induction hypothesis there are ρ_H, π_H such that (H, ρ_H, π_H) is a map and

$$\text{genus } (H, \rho_H, \pi_H) \leq \text{genus } (G - \{a, a'\}, \pi_{G-\{a,a'\}}, \rho_{G-\{a,a'\}}) \leq \text{genus } (G, \pi_G, \rho_G).$$

Case $\delta_G \neq 0$ *and* $|\mathsf{A}_H| = |\mathsf{A}_G|$*:* Then there is a $v \in \mathsf{V}_G \setminus \mathsf{V}_H$ and v is isolated in G. By Lemma 4.45 and the induction hypothesis, the property holds. □

Corollary 4.57. *Let* (G, π_G, ρ_G) *be a map. Then* $0 < \text{genus } (G, \pi_G, \rho_G)$.

Proof. The empty graph has genus 0 and is a subgraph of G. The corollary then follows by Lemma 4.56. □

Corollary 4.58. *Let* G *be a combinatorially planar graph and* H *a bidirectable subgraph of* G*. Then* H *is combinatorially planar.*

Proof. Follows from Lemma 4.56 and Corollary 4.57. □

With this, we can finally prove that combinatorial and Kuratowski planarity are compatible:

Theorem 4.59. *Let* G *be a combinatorially planar graph. Then* G *is also Kuratowski-planar.*

Proof. By contradiction: assume that G is not Kuratowski-planar. Then G has a subgraph H which is a subdivision of some Kuratowski subgraph K. By Corollary 4.58 is H combinatorially planar and hence by Theorem 4.44 also K.

On the other hand, K is not combinatorially planar by Corollary 4.34. This is a contradiction and hence the assumption that G is not Kuratowski-planar is false. □

4.7. Discussion

In this chapter, I formalized two characterizations of planarity: a combinatorial one using planar maps and one based on Kuratowski's theorem. These two characterizations were chosen as planar maps respectively Kuratowski subgraphs can be used to easily check that a graph is planar (w.r.t. combinatorial planarity) respectively not planar (w.r.t. Kuratowski planarity). Imperative implementations of these checks are verified in Chapter 5 and Chapter 7.

Furthermore, I proved that combinatorial planarity implies Kuratowski-planarity, so that both planarity and non-planarity can be certified with regard to the same definition. As a side effect of this proof, I presented a decision procedure for combinatorial planarity. This procedure is terribly inefficient: for a star graph with n vertices[3], the algorithm enumerates $(n-2)!$ different maps. An efficient algorithm can decide the problem in linear time (cf. Mohar and Thomassen [55, §2.7]). Nevertheless, the algorithm is efficient enough to decide planarity for small graphs like the K_5 or $K_{3,3}$.

Usually, planarity is considered for undirected graphs. Nevertheless, I decided to base the definitions on directed graphs, which allows me to use the theory of digraphs formalized in Chapter 3. This choice is easily justified for combinatorial planarity, which is described in terms of half-edges in the literature (e.g. Lando and Zvonkin [44]). For the characterization based on Kuratowski's theorem, the used vocabulary is the standard one for digraphs, except for the definition of subdivision (for digraphs, one usually considers subdividing a single arc, not a pair of arcs). For the results formalized in this chapter, the definition based on digraphs works well: these are concerned with the relation to combinatorial planarity, which is based on bidirected digraphs anyway. In Chapter 5, I prove further properties about Kuratowski planarity, which gives further insights whether the use of digraphs is suitable here.

Planarity has been the topic of a number of formalizations. In their proof of the Five Color Theorem, Bauer and Nipkow [9] define planar graphs in Isabelle/HOL inductively in terms of near triangulations. Their inductive definition has been motivated by Yamamoto et al. [84], who used a similar definition to prove the Euler equation.

Gonthier [31] formalized a proof of the Four Color Theorem. Most of the proof is done on planar hypermaps, a generalization of the concept of planar maps. As part of this work, he also proves a version of the Jordan Curve Theorem for hypermaps, which gives a new combinatorial characterization of planarity. Hypermaps have also been the basis of other formal definitions of planarity. Dufourd [20] used hypermaps to formalize a discrete version of the Jordan Curve Theorem. The proof of the Kepler conjecture by [34] uses the concept of tame plane graphs. The formalization of this proof in the Flyspeck project [35] defines these twice, both inductively [57] and based on hypermaps.

The formalizations mentioned above only consider planar graphs in a specialized setting. My contribution is the formalization of planarity in the context of a wider graph library, using the standard structures of said library, and a partial proof of equivalence between the two formalized characterizations of planarity.

This proof development takes around 8000 lines of Isabelle proof text. More than a third of this relates to Section 4.3 and Section 4.4. A fourth of this is specific to Section 4.6 and the

[3] I.e., $G = (\{1, \ldots, n\}, \{(u, v) \mid 1 \leq u, v \leq n \wedge (u = 1 \vee v = 1)\})$

remainder is roughly evenly split between Section 4.5 and lemmas used in all three parts. If one trusts the basic vocabulary of Isabelle/HOL, the vocabulary necessary for combinatorial planarity consists of around 50 lines of definitions.

5. A Checker for Non-Planarity

In this chapter, I verify a checker for non-planarity of graphs. Such a checker is a program which takes a graph and a certificate (in this case, a Kuratowski subgraph) and decides whether the certificate proves that the graph is non-planar. I verify two imperative implementations of the same algorithm and discuss the differences in effort needed. One version is implemented in an abstract language using standard Isabelle expressions and the other one in C.

For many applications, it is important that a program only computes correct results. One way to achieve this is the formal verification of the program. Unfortunately, this is not always feasible. Generally, verifying a program is an order of magnitude harder then implementing it. Experience shows that for the verification of complex programs, it is crucial to keep verification in mind during the implementation. In particular, this means avoiding constructs which are hard to verify.

Moreover, "ease of verification" and "efficiency of implementation" are often on opposite ends of the scale. This holds both for optimizations in the implementation and in the algorithm.

If we can tolerate a wrong result, as long as the error is immediately noticed, another approach becomes viable. A *certifying algorithm* is an algorithm which takes an input x and returns both a result y and a *witness* w. An accompanying *checker* then uses w to ascertain that y is the correct result for input x. A small example helps understanding this concept: the input of a planarity test is a graph, the output is "planar" or "not planar". A certifying planarity test might witness the result "planar" by a plane map and the result "not planar" by a Kuratowski subgraph. The checker would then check that the map is plane respectively that the graph is a Kuratowski subgraph of the input. If the check passes, the result is correct. Otherwise it might be wrong and should therefore be rejected.

The idea of a checker for the output of a program was described by Blum and Kannan [12] and Sullivan and Masson [73] developed the idea of certification trails. Certifying algorithms [50] generate easily checkable certificates and are a key design principle of the algorithms library LEDA [52]. Checkers are an integral part of the library and are optionally invoked after every execution of a LEDA program. Adoption of this principle greatly improved the reliability of the library [51].

Usually, a checker is a much simpler program than the algorithm it belongs to. This makes checkers amenable to formal verification. Note that one checker can be used for a whole class of algorithms, as long as all of these produce the same certificates: this allows optimizing the implementation or even replacing it by a completely different algorithm without redoing the verification. Recently, Alkassar et al. [2] developed a framework for verifying certifying computations. Their approach combines the automatic code verifier VCC [17] and the interactive theorem prover Isabelle to prove correctness of checkers: VCC is used to prove low-level properties of the C-code. These low-level properties are then used in Isabelle to derive the desired mathematical properties, which are then translated back to VCC. In this approach, each

of the two proof tools is used according to its strengths.

Another approach is to perform the whole verification in Isabelle. This greatly reduces the amount of trusted code and allows for a richer language for the specification and eliminates the need for transferring theorems between two different logics. This *Isabelle approach* has been discussed by Noschinski, Rizkallah, and Mehlhorn [67]. In this chapter, I follow the Isabelle approach to verify a checker for the non-planarity of graphs. To compare the work needed for an imperative formulation of the checker algorithm and an actual implementation, I implemented the checker twice, in Simpl and in C. Simpl [71] is a generic imperative language embedded into Isabelle/HOL, designed as an intermediate language for verification. It allows using arbitrary Isabelle datatypes and expressions in programs. To translate from C to Isabelle, I used the AutoCorres tool by Greenaway, Andronick, and Klein [33]. This builds on a C-to-Isabelle parser [76] and simplifies the output by automatic abstraction. The hope was that after completing the Simpl verification, the verification of the C program would mainly consist of dealing with intricacies of C.

This chapter is based on a paper published in the proceedings of the NASA formal methods workshop 2014 [67]. Introduction, the following section and the comparison of different verification approaches are joint work with Christine Rizkallah and Kurt Mehlhorn. The formalization of the checker algorithm, its implementation in C and Simpl, and the verification of the implementations are my own work.

5.1. Certifying Algorithms

Following the framework defined by Alkassar et al. [2], a certifying algorithm takes an input x from a set X and produces an output y from a set Y and a witness w from a set W. Input x is supposed to satisfy a precondition $\varphi\ x$, and x and y are supposed to satisfy a postcondition $\psi\ (x, y)$. A *witness predicate* for a specification with precondition φ and postcondition ψ is a predicate $\mathcal{W} \subseteq X \times Y \times W$ with the following *witness property*:

$$\forall x, y, w.\ \varphi\ x \wedge \mathcal{W}\ (x, y, w) \implies \psi\ (x, y) \tag{5.1}$$

In contrast to algorithms, which work on abstract sets X, Y, and W, programs as their implementations operate on concrete representations of abstract objects. I use $\overline{X}$, $\overline{Y}$, and $\overline{W}$ for the set of representations of objects in X, Y, and W, respectively and assume mappings $i_X : \overline{X} \to X$, $i_Y : \overline{Y} \to Y$, and $i_W : \overline{W} \to W$. The checker program C receives a triple $(\overline{x}, \overline{y}, \overline{w})$ and is supposed to check whether this triple fulfills the witness property. More precisely, let $x = i_X\ \overline{x}$, $y = i_Y\ \overline{y}$, and $w = i_W\ \overline{w}$. If $\neg\varphi\ x$, C may do anything (run forever or halt with an arbitrary output). If $\varphi\ x$, C must halt and either accept or reject. It is supposed to accept if $\mathcal{W}\ (x, y, w)$ holds and to reject otherwise. The following proof obligations arise:

Witness Property: A proof for the implication (5.1).

Checker Correctness: A proof that C checks the witness predicate if the precondition φ is satisfied. I.e., for an input $(\overline{x}, \overline{y}, \overline{w})$ with $x = i_X\ \overline{x}$, $y = i_Y\ \overline{y}$, $w = i_W\ \overline{w}$:

1. If $\varphi\ x$, C halts.

2. If $\varphi\ x$ and $\mathcal{W}\ (x, y, w)$, C accepts, and if $\varphi\ x$ and $\neg\mathcal{W}\ (x, y, w)$, C rejects.

Consider the planarity test from the LEDA library [52]. This algorithm takes as input a graph x and returns $y = \mathsf{True}$ and a plane map w of x if x is planar or $y = \mathsf{False}$ and a Kuratowski subgraph w of x otherwise. Therefore, the framework is instantiated as follows.

input x	=	undirected graph, represented as bidirectable digraph G[1]
output y	=	either True or False
witness w	=	plane map or Kuratowski subgraph
$\varphi\ x$	=	G is wellformed and V_G and A_G are finite.
$\psi\ (x, y)$	=	If y is True, x is combinatorially planar, else x is not combinatorially planar.

A checker for this algorithm needs to check two kinds of different certificates, so I split $\mathcal{W}$ into two separate predicates $\mathcal{W} = \mathcal{W}_{\mathsf{True}} \cup \mathcal{W}_{\mathsf{False}}$ where $\mathcal{W}_{\mathsf{True}}$ and $\mathcal{W}_{\mathsf{False}}$ are the witness predicates for $y = \mathsf{True}$ respectively $y = \mathsf{False}$. The description above suggests the following definitions of the witness predicate:

$$\mathcal{W}_{\mathsf{True}} := \{(x, y, w) \mid y = \mathsf{True} \wedge w \text{ is a planar map for } x\}$$
$$\mathcal{W}_{\mathsf{False}} := \{(x, y, w) \mid y = \mathsf{False} \wedge w \text{ is a Kuratowski subgraph of } x\}$$

The LEDA library contains checkers for both the planarity and the non-planarity case. In this thesis, I will present verified implementations for both cases, based on the implementations in LEDA.

For the case $y = \mathsf{False}$, the implementation in LEDA accepts also graphs which are only "almost" Kuratowski subgraphs. In Section 5.2 I consider the algorithm used in this implementation, prove its correctness and give an updated definition of $\mathcal{W}_{\mathsf{False}}$. The verification of an implementation in Simpl is covered in Section 5.3, the verification of an implementation in C in Section 5.4. After introducing a tool for easier verification in Chapter 6, the case for $y = \mathsf{True}$ is discussed in Chapter 7.

5.2. A Checker Algorithm for Non-Planarity

Given the description of the certifying algorithm in the previous section, the checker for the non-planarity predicate $\mathcal{W}_{\mathsf{False}}$ must decide whether the witness is a subgraph of the input and a subdivision of a Kuratowski graph. To decide the latter property, the implementation of this checker in the LEDA library contracts the graph and accepts the witness if the result is a Kuratowski graph. In this section, I analyze the contraction algorithm and prove that it is suitable for checking non-planarity. I also give an exact characterization of the class of accepted witnesses.

In this section undirected graphs are modeled as finite and bidirectable digraphs, using the pair graph representation from Section 3.2.1. Unless otherwise specified, the term graph refers to such digraphs in this section. The term *edge* refers to an arc and its reverse arc.

[1] The implementation of the planarity test in LEDA accepts any digraph as input and interprets it as an undirected graph. However, a planarity certificate is only computed for bidirected digraphs.

Subdivision was defined in Chapter 4. For pair graphs, the following simpler definition is equivalent.

Lemma 5.1 (Subdivision on pair graphs). *A subdivision splits an edge (u, v) by inserting a new node w. For a pair graph G, the graph* subdivide G (u, v) w *is defined as*

$$(\mathsf{V}_G \cup \{w\}, \mathsf{A}_G \setminus \{(u, v), (v, u)\} \cup \{(u, w), (w, u), (v, w), (w, v)\}).$$

Then, for two pair graphs, subdivision is characterized by the following inductive definition:

$$[\![\text{bidirected-digraph } G]\!] \implies \text{subdivision } G\ G$$
$$[\![a \in \mathsf{A}_G;\ w \notin \mathsf{V}_G;\ \text{subdivision } G\ H]\!] \implies \text{subdivision } G\ (\text{subdivide } H\ a\ w)$$

Vertices added by subdivision always have degree 2. A $\mathsf{K}_{3,3}$ or K_5 has no vertices of degree 2.

So, to check whether a graph G is a subdivision of a Kuratowski graph, one can undo the subdivision step-by-step by contracting those vertices of G which have degree 2.

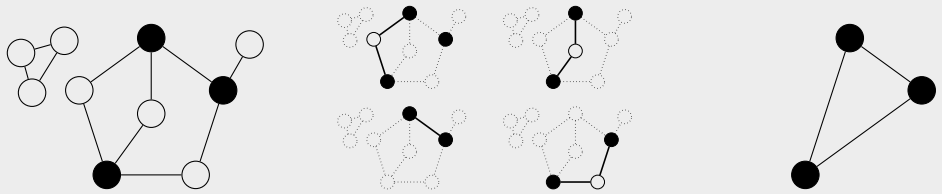
To avoid having to construct the intermediate graphs, the LEDA checker procedure approximates this process by contracting all these vertices at once.

Definition 5.2 (Restricted Walk). *The* inner vertices *of a walk are the vertices of the walk, excluding the endpoints. A walk is* progressing*, if it does not contain the sequence $[(u, v), (v, u)]$ for any vertices u, v. Moreover, a* restricted walk *w.r.t. V, is a non-empty walk which is progressing and has no inner vertices in V. Finally, a* restricted path *w.r.t. V is a path which is a restricted walk w.r.t. V and has endpoints in V (note that any path is progressing).*

$$\text{inner-verts}_G\ p \coloneqq \text{tl}\ (\text{hd}\ (\text{aw-verts}_G\ p))$$
$$\text{progressing}\ p \coloneqq \forall xs, x, y, ys.\ p \neq xs +\!\!+ [(x, y), (y, x)] +\!\!+ ys$$
$$\text{rawalk}_G\ V\ u\ p\ v \coloneqq \text{awalk}_G\ u\ p\ v \wedge p \neq [] \wedge \text{set}\ (\text{inner-verts}_G\ p) \cap V = \varnothing \wedge \text{progressing}\ p$$
$$\text{rapath}_G\ V\ u\ p\ v \coloneqq \text{apath}_G\ u\ p\ v \wedge p \neq [] \wedge \{u, v\} \subseteq V \wedge \text{set}\ (\text{inner-verts}_G\ p) \cap V = \varnothing$$

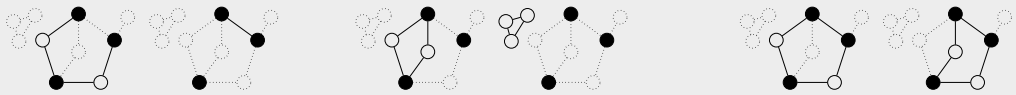
A progressing walk cannot alternate back and forth between to vertices, so such a walk does not contain any cycles (but it might be a cycle, if the end vertices are equal). If G is a subdivision of H, a walk for H can be transformed into a walk for G by subdividing is edges and a walk for G can be transformed into a walk for H by contracting its edges (as long as the end vertices exist in both graphs). The formalization contains functions for single-step subdivision and contraction of walks. Many properties, in particular the property of being a restricted path, a preserved under these transformations. The proofs for that are technical and not very illuminating. Nevertheless, a considerable fraction of the work which went into the formalization of this section was spent on these.

In my formalization, V always satisfies $\mathsf{V}_G \subseteq V \subseteq \mathsf{V}_{3,G} \subseteq V$, where $\mathsf{V}_{3,G} \coloneqq \{v \in \mathsf{V}_G.\ 3 \leq \text{in-deg}_G\ v\}$ denotes the set of vertices of G of degree 3 or more. Then the inner vertices of a restricted walk have degree 2, i.e., the walk could have been constructed by repeated subdivision of a single arc. In a way, restricted paths w.r.t. V describe the edges of a graph with verts V. Therefore, the arcs of the *contracted graph* contr-graph G V are defined by the existence of an restricted path.



(a) G (b) all restricted paths (c) contracted graph

Figure 5.1.: A graph G and its restricted paths and contracted graph w.r.t. $V_{3,G}$ (in black). Neither the separate cycle nor the node of degree 1 are on any restricted path (or in $V_{3,G}$), so they do not contribute to the contracted graph.



(a) (b) (c)

Figure 5.2.: Some slim and non-slim subgraphs of G from Figure 5.1a. (a) shows some slim subgraphs w.r.t. $V_{3,G}$ (in black). The two graphs depicted in (b) are not slim: the first one has parallel restricted paths, the second one contains vertices not on a restricted path. One of the two graphs in (c) is arbitrarily chosen as the slim graph of G.

Definition 5.3 (Checker Procedure).

$$\text{contr-graph}\ G\ V \coloneqq (V, \{(u,v) \mid \exists p.\ \text{rapath}_G\ V\ u\ p\ v\})$$
$$\text{certify}\ G\ C \coloneqq \text{subgraph}\ C\ G \wedge (\text{let}\ H = \text{contr-graph}\ C\ V_{3,C}\ \text{in}\ (K_{3,3}\ H \vee K_5\ H))$$

Here, V is the set of vertices to be preserved by the contraction.

Figure 5.1 illustrates these definitions. It is easy to see that this contraction is not an exact inverse to subdivision: vertices of degree 1 or 0, separate cycles or parallel restricted paths are removed, even though they cannot be constructed by subdivision.

On the other hand, if G is a subdivision of H and all vertices of H have degree 3 or more then the contraction reverses the subdivision and hence certify accepts all Kuratowski subgraphs as certificate.

Lemma 5.4. *Let G, H be graphs such that H is loop-free and G is a subdivision of H. If all vertices of H have degree 3 or more, then* $\text{contr-graph}\ G\ V_{3,G} = H$.

Proof. Straightforward induction over the subdivision. □

Theorem 5.5 (Completeness). *Let G, H be graphs. If H is a Kuratowski subgraph of G, then the predicate* $\text{certify}\ G\ H$ *holds.*

$$[\![\text{subgraph}\ H\ G;\ \exists H'.\ (K_{3,3}\ H' \vee K_5\ H') \wedge \text{subdivision}\ H'\ H]\!] \implies \text{certify}\ G\ H$$

Proof. Follows directly from Lemma 5.4, as both $K_{3,3}$ and K_5 have only vertices of degree 3 or higher. □

Even though the contraction does more then reversing the effects of a subdivision, the procedure is still suitable to guarantee that a graph is non-planar: if a graph does not contain any of the features mentioned above, it is a subdivision of its contracted graph by Lemma 5.4. Even for an arbitrary graph H, there is always a subgraph H' without these features, such that H' is a subdivision of contracted graph of H. Such graphs are called *slim*.

Definition 5.6 (Slim Graph). *Intuitively, a digraph is* slim *if it is minimal in the sense that removing a vertex or an arc would lead to a smaller contracted graph. This means that all vertices and arcs lie on restricted paths and there is at most one restricted path connecting each pair of vertices.*

$$\begin{aligned}\text{is-slim}_G\ V \coloneqq \big(&V \subseteq \mathsf{V}_G \wedge (\forall v \in \mathsf{V}_G.\ v \in V \vee (\text{in-deg}_G\ v < 3 \\ &\wedge (\exists x, p, y.\ \text{rapath}_G\ V\ x\ p\ y \wedge v \in (\text{aw-verts}_G\ x\ p)))) \\ &\wedge (\forall a \in \mathsf{A}_G.\ \exists x, p, y.\ \text{rapath}_G\ V\ x\ p\ y \wedge a \in p) \\ &\wedge (\forall x, y, p, q.\ (\text{rapath}_G\ V\ x\ p\ y \wedge \text{rapath}_G\ V\ x\ q\ y) \Longrightarrow p = q)\big)\end{aligned}$$

Moreover, for a bidirectable digraph the slim graph of G *is defined as follows. Let*

$$R \coloneqq \{(u, v) \mid \exists p.\ \text{rapath}_G\ \mathsf{V}_{3,G}\ u\ \ p\ v\}$$

be the set of all pairs of vertices connected by a restricted path. Let ch-p *be a function mapping* (u, v) *to a restricted path from* u *to* v*, such that the edges of* ch-p (u, v) *are the reversed edges of* ch-p (v, u) *in reverse order. Then* slim G *is defined as the graph consisting of all the vertices and arcs in* ch-p‘R.

These definitions are illustrated in Figure 5.2. The idea of slim G is to remove just the parts of G which make G non-slim. Proving that slim G is a slim subgraph of G w.r.t $\mathsf{V}_{3,G}$ is straightforward, except for the proof that there are no parallel restricted paths in slim G.[2] For this, we need to know that a restricted path is uniquely determined by any of its arcs. This follows from the following lemma.

Lemma 5.7 (Unique Restricted Walks). *Let V be a set of vertices with $\mathsf{V}_{3,G} \subseteq V \subseteq \mathsf{V}_G$. Moreover, let* $\text{rawalk}_G\ V\ u\ p\ v$ *and* $\text{rawalk}_G\ V\ w\ q\ x$ *with $\{u, v, w, x\} \subseteq V$, $a \in p$ and $a \in q$. Then $p = q$.*

Proof. Intuitively this is clear, as a restricted walk ending in a vertex of degree 2 can only be extended by a uniquely defined arc.

For a formal proof, note that p is of the form $p_0 +\!\!+ [a] +\!\!+ p_1$ and q of the form $q_0 +\!\!+ [a] +\!\!+ q_1$. We now prove that $p_0 +\!\!+ [a] = q_0 +\!\!+ [a]$ and $[a] +\!\!+ p_1 = [a] +\!\!+ q_1$. Then the $p = q$ follows.

The two cases are symmetric (by reversing the walks), so proving the second case suffices.

[2] If one is only interested in the correctness of the checker, there is no need to introduce the concept of slim graphs. However, slim graphs allow us to give a more explicit characterization than “the contracted graph is a kuratowski graph”.

We prove the following proposition for arbitrary u, v, x, p, and q by parallel induction on p and q: Let rawalk$_G$ V u p v and rawalk$_G$ V u q x such that $p, q \neq []$ and hd p = hd q. If $v, x \in V$, then $p = q$ holds . This can obviously be used to discharge the second case above.

Due to $p, q \neq []$ and hd p = hd q, it suffices to consider the case with $p = (u, u') :: as$ and $q = (u, u') :: bs$. We have $as = [] \iff bs = []$ (because otherwise we have $u' \in V$ and $u' \in$ inner-verts$_G$ $p \cup$ inner-verts$_G$ q). For $as = bs = []$ we are done.

Otherwise, we can assume that

$$p = (u, u') :: as = (u, u') :: (u', u_a) :: as' \qquad p = (u, u') :: as = (u, u') :: (u', u_b) :: bs'$$

for some vertices u_a, u_b and lists as' and bs'. As p, q are progressing, we have $u \neq u_a$ and $u \neq u_b$. As u' is an inner vertex, $u' \notin V$ and hence u' has a degree of at most 2. Therefore, $u_a = u_b$. As as and bs are again restricted walks, we can now apply the induction hypothesis to prove that $as = bs$ and with $a = b$ follows $p = q$. □

With the uniqueness of restricted paths, it is easy to show that slim G contains no parallel restricted paths.

Lemma 5.8. *The graph* slim G *is slim w.r.t* $V_{3,G}$.

Proof. All properties except that there are no parallel restricted paths are easy to show.

Assume that there are two different restricted paths rapath x p y and rapath x q y in slim G. These are also restricted paths in G.

Let R be defined as in Definition 5.6. As a restricted path, p is not empty and shares an edge with a restricted path $p' \in$ ch-p R. By the uniqueness of paths (Lemma 5.7), $p = p'$ and hence $p =$ ch-p (x, y). Similarly, $q =$ ch-p (x, y) and therefore $p = q$. □

A slim graph is indeed always a subdivision of its contracted graph:

Lemma 5.9.

$$[\![\text{is-slim}_G\ V]\!] \implies \text{subdivision (contr-graph } G\ V)\ G$$

Proof. We prove this by induction on $|V_G \setminus V|$. The proof of the base case follows directly from contr-graph G $V_G = G$ for slim graphs G.

For the induction step, from G being slim we obtain a $w \in V_G \setminus V$, which is an inner vertex of some restricted path p. Hence p contains the arc sequence $(u, w), (w, v)$ where u, v are the only two neighbors of w in G. As G is slim, $(u, v) \notin A_G$: otherwise we could replace $(u, w), (w, v)$ by (u, v) in p and get a restricted path parallel to p. From this, we can construct a graph H such that $G =$ subdivide H (u, v) w. Note that the subdivide operation does not change the contracted graph.

As w does not occur in H, the inequality $|V_H \setminus V| < |V_G \setminus V|$ holds. As G is slim and a subdivision of H, H is also slim. Then subdivision (contr-graph H V) H holds from the induction hypothesis and by the transitivity of subdivision follows subdivision (contr-graph G V) G. □

In particular, this already implies that the checker procedure is sound for slim certificates. For a non-slim certificate C, we make use of the fact that C and slim C have the same contracted graph w.r.t. $V_{3,C}$.

Lemma 5.10.

$$\text{contr-graph } (\text{slim } G)\ \mathsf{V}_{3,G} = \text{contr-graph } G\ \mathsf{V}_{3,G}$$

Proof. Follows as both G and slim G have the same nodes connected by restricted paths. □

As the slim graph of C is a subgraph of C, the checker is sound.

Theorem 5.11 (Soundness). *Let G, C be graphs. If* certify $G\ C$ *holds, then G is not Kuratowski-planar.*

Proof. From the assumption, contr-graph $C\ \mathsf{V}_{3,C}$ is a Kuratowski graph. By Lemma 5.10, we have contr-graph $C\ \mathsf{V}_{3,C}$ = contr-graph (slim C) $\mathsf{V}_{3,C}$ and, with Lemma 5.9, slim C is a subdivision of contr-graph $C\ \mathsf{V}_{3,C}$. In addition slim C is a subgraph of C and from the assumption C is a subgraph of G. Hence, G has a subgraph, namely slim C, which is a subdivision of a Kuratowski graph and hence G is not Kuratowski-planar. □

From Theorem 5.5 and Theorem 5.11 follows that certify is a sound and complete checker. As demonstrated by Figure 5.1, not only Kuratowski subgraphs are accepted as certificates. Based on this, I redefine the witness predicate $\mathcal{W}_{\mathsf{False}}$ as follows:

$$\begin{aligned}\mathcal{W}_{\mathsf{False}} \coloneqq \{(x, y, w) \mid\ & y = \mathsf{False} \wedge w \text{ is wellformed and loop-free} \\ & \wedge \text{ contr-graph } w\ \mathsf{V}_{3,w} \text{ is Kuratowski graph}\}\end{aligned}$$

In theory, there is no need to exclude certificates with loops. However, the implementations of certify presented in the following sections use this to simplify the computation of the contracted graph.

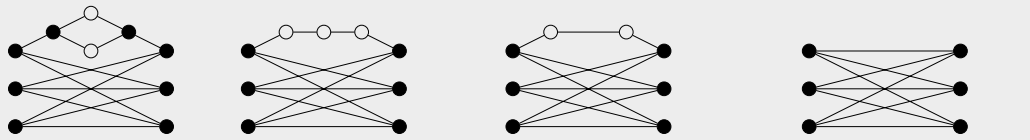
I conclude this section with a more concrete characterization of the accepted certificates. Looking at the proof of Theorem 5.11, we see that a certificate is accepted only if its slim graph is a subdivision of a Kuratowski graph. As illustrated in Figure 5.3, this is not a sufficient condition: in addition, the slim graph must have the same vertices of degree 3 or more as the original graph.

Theorem 5.12 (Characterization of the Accepted Certificates). *For a graph G, the function* certify *accepts exactly those graphs C as a certificate, which are subgraphs of G and have a slim graph which is a subdivision of a Kuratowski graph. Moreover, the slim graph must have the same vertices of degree 3 or more as C itself.*

$$\begin{aligned}\text{certify } G\ C \iff\ & \text{subgraph } C\ G \wedge \mathsf{V}_{3,C} = \mathsf{V}_{3,\text{slim } C} \\ & \wedge\ (\exists H.\ (\mathsf{K}_{3,3}\ H \vee \mathsf{K}_5\ H) \wedge \text{subdivision } H\ (\text{slim } C))\end{aligned}$$

Proof. Assume that certify $G\ C$ holds. Then C is a subgraph of G by the definition of certify and $D \coloneqq$ contr-graph $C\ \mathsf{V}_{3,C}$ is a Kuratowski graph.

We first note that $\mathsf{V}_{3,\text{slim } C} = \mathsf{V}_{3,C}$ holds: By Lemma 5.10, the contracted graph of slim C w.r.t. $\mathsf{V}_{3,C}$ is the same as D. The vertices of D are $\mathsf{V}_{3,C}$ and, as D is a Kuratowski graph, all of them have degree 3 or more in D. As the degree of a vertex in the contracted graph is less or equal to



(a) G (b) slim G (c) contr-graph G $V_{3,G}$ (d) contr-graph G $V_{3,\mathsf{slim}\ G}$

Figure 5.3.: The black nodes are vertices of degree 3. The slim graph of G is obviously a subdivision of a $K_{3,3}$. Still, the contracted graph of G w.r.t. $V_{3,G}$ is not a $K_{3,3}$ (but the contracted graph w.r.t. $V_{3,\mathsf{slim}\ G}$ is).

the degree of the same vertex in the original graph, this implies $V_{3,C} \subseteq V_{3,\mathsf{slim}\ C}$. As slim C is a subgraph of C, we have $V_{3,\mathsf{slim}\ C} \subseteq V_{3,C}$ and therefore $V_{3,\mathsf{slim}\ C} = V_{3,C}$.

Hence, D is the contracted graph of slim C w.r.t. $V_{3,\mathsf{slim}\ C}$. Then, by Lemma 5.9, slim C is a subdivision of D and thus the right hand side of the characterization follows.

For the other direction, it suffices to show that the contracted graph of C w.r.t. $V_{3,C}$ is a Kuratowski graph. As the contraction of a subdivision of a Kuratowski graph is again said Kuratowski graph (cf. Lemma 5.4), the contracted graph of slim C w.r.t. $V_{3,\mathsf{slim}\ C}$ is a Kuratowski graph. As $V_{3,C} = V_{3,\mathsf{slim}\ C}$, this is the same as the contracted graph of C w.r.t. $V_{3,C}$ and we are finished. □

5.3. Implementation in Simpl

In the previous section, I defined the witness predicate $\mathcal{W}_{\mathsf{False}}$ and showed that it fulfills the witness property. As discussed before, I will present two implementations for computing $\mathcal{W}_{\mathsf{False}}$. I start with the implementation in Simpl, discussing the algorithm and the key steps of verification.

Simpl is an imperative language designed for program verification. It has the usual control structures: conditional statements (**IF** ... **THEN** ... **FI**), loops (**WHILE** ... **DO** ... **OD**) and exceptions (**RAISE** and **TRY** ... **CATCH** ... **END**). Expressions are arbitrary Isabelle expressions. These can refer to program variables, which can be imperatively updated by assignment (... := ...). Like constants, program variables are typeset in sans-serif.

The implementation of the checker is roughly divided in four steps. Given a graph G and a witness graph C, the implementation

1. tests whether C is a subgraph of G,
2. tests whether C is loop free,
3. computes the contracted graph of C,
4. and tests whether the contracted graph is a Kuratowski graph.

```
type_synonym IVert = ℕ
type_synonym IEdge = IVert × IVert
record IGraph = ig-vs : IVert list, ig-es : IEdge list

definition ig-wf : IGraph → bool where
  ig-wf G ≔ distinct (ig-vs G) ∧ (∀(u, v) ∈ ig-es G. u ∈ ig-vs G ∧ v ∈ ig-vs G)
definition ig-adj : IVert → IVert → bool where
  ig-adj G u v ≔ (u, v) ∈ ig-es G ∨ (v, u) ∈ ig-es G
definition ig-in-out-edges : IGraph → IVert → IEdge list where
  ig-in-out-edges G u ≔ filter (λe. fst e = u ∨ snd e = u) (ig-es G)
definition ig-opposite : IGraph → IEdge → IVert → IVert where
  ig-opposite G e u ≔ (if fst e = u then snd e else fst e)
```

Figure 5.4.: Excerpt from the specification of the graph type in the Simpl program. ig-wf is the wellformedness condition. The predicate ig-adj describes the adjacency relation, ig-in-out-edges gives a list of incident edges, and ig-opposite returns the other vertex of an edge.

The input is accepted if and only if all these tests succeed. The most instructive part is the verification of the contraction, so I focus on step 3. Nevertheless, I verified the full algorithm in Isabelle.

The graphs in this section are again undirected graphs. I use two different representations: On the abstract side, graphs are bidirected pair graphs, as in the previous section. In the Simpl program, we need a way of iterating over arcs and vertices. A convenient way to achieve this is to use lists instead of sets. Therefore graphs are represented as a list of vertices and a list of (undirected) edges. An edge is represented as a pair of vertices, i.e., (u, v) and (v, u) describe the same edge.

For the implementation, I assume that I already have a correct implementation of basic graph operations. This is achieved by using Isabelle definitions instead of Simpl procedures for these operations. The graph type and its operations are given in Figure 5.4.

The code to compute the contracted graph is split in three parts: First, a graph H with no edges is computed by taking all vertices of degree three or more of C. The core of the computation is then performed by the function find-endpoint (Figure 5.5): For a given edge $(\mathsf{v}_{\mathsf{start}}, \mathsf{v}_{\mathsf{next}})$ it computes the restricted path of G starting with this edge end returns its last vertex (if it exists). The edge corresponding to this restricted path is then added to H. This step is repeated for all edges of G incident to some vertex in H.

5.3.1. Verification

The specification of find-endpoint is given in Figure 5.6. I now outline the proof of this specification. The function contains two nested loops: the outer loop constructs an restricted walk

```
procedures find-endpoint (G : IGraph,
   H : IGraph,  v_start : IVert, v_next : IVert
   | R : IVert option)
where
 found : bool, i : ℕ, len : ℕ, v_0 : IVert,
 v_1 : IVert, v_t : IVert, io-edges : IEdge list,
TRY
 IF v_start = v_next THEN
  RAISE R := None
 FI ;;
 v_0 := v_start ;; v_1 := v_next ;; len := 1 ;;
 WHILE v_1 ∉ ig-vs H DO
  io-edges := ig-in-out-edges G v_1 ;;
  i := 0 ;; found := False ;;
  WHILE ¬found ∧ i < |io-edges| DO
   v_t := ig-opposite G (io-edges ! i) v_1 ;;
   IF v_t ≠ v_0 THEN
    found := True ;; v_0 := v_1 ;; v_1 := v_t
   FI ;;
   i := i + 1
  OD ;;
  len := len + 1 ;;
  IF ¬ found THEN RAISE R := None FI
 OD ;;
 IF v_1 = v_start THEN
  RAISE R := None
 FI ;;
 R := Some v_1
CATCH SKIP END
```

```
unsigned find_endpoint(
  struct graph_t *g,
  struct contr_t *h,
  unsigned v_start, unsigned v_next)
{
  unsigned v0 = v_start;
  unsigned v1 = v_next;

  while (tmp_get_index(h, v1) == -1) {
    unsigned i;
    for (i=0; i < edge_cnt(g); i++) {
      unsigned vt;
      vt = opposite(v1, edge(g,i));
      if (vt != v0 && vt != -1) {
        v0 = v1;
        v1 = vt;
        break;
      }
    }
    if (i == edge_cnt(g)) return -1;
  }
  if (v1 == v_start) return -1;
  return v1;
}
```

Figure 5.5.: The function find-endpoint in Simpl and C. H (resp. h) contains the vertices of degree 3 or more. The function (implicitly) constructs an restricted path starting with the edge (v_{start}, v_{next}) by adding edges until a vertex in H is reached. The conditional statement in the inner loop ensures that the computed walk is progressing. If the outer loop aborts abnormally, then no vertex in H is reachable from v_{start} via (v_{start}, v_{next}).
The Simpl implementation uses relatively high-level datastructures, like sets and lists, the C implementation uses arrays.

$$\begin{aligned}
&\forall\sigma.\ \Gamma \vdash_t \{\!|\sigma.\ \text{ig-vs H} = \text{ig-vs}_3\ \text{G} \wedge \text{loop-free (mk-graph G)} \wedge \text{v}_{\text{start}} \in \text{ig-vs H} \\
&\quad \wedge \text{iadj G v}_{\text{start}}\ \text{v}_{\text{next}} \wedge \text{ig-wf G}|\!\} \\
&\quad \text{R} := \textbf{PROC}\ \text{find-endpoint(G, H, v}_{\text{start}}, \text{v}_{\text{next}}) \\
&\{\!|\text{case R of None} \Rightarrow \neg(\exists p\ w.\ \text{rapath}_{\text{mk-graph G}^\sigma}\ \text{v}^\sigma_{\text{start}}\ ((\text{v}^\sigma_{\text{start}}, \text{v}^\sigma_{\text{next}}) :: p)\ w) \\
&\qquad\qquad | \text{Some } w \Rightarrow (\exists p.\ \text{rapath}_{\text{mk-graph G}^\sigma}\ \text{v}^\sigma_{\text{start}}\ ((\text{v}^\sigma_{\text{start}}, \text{v}^\sigma_{\text{next}}) :: p)\ w)
\end{aligned}$$

Figure 5.6.: Specification of find-endpoint: If H has all degree-3 nodes of G and G has no loops, then the procedure decides the existence of a restricted path starting with the nodes v_{start} and v_{next}. The function mk-graph abstracts a graph and x^σ refers to the value of x before the execution.

by adding an edge per iteration; the inner loop selects the next edge which with the path is extended (if such an edge exists).

The key part of the invariant of the outer loop is the following formula (the variables correspond to those in Figure 5.5):

$$\exists p.\ \text{rawalk}_G\ \text{v}_{\text{start}}\ p\ \text{v}_1 \wedge |p| = \text{len} \wedge \text{hd}\ p = (\text{v}_{\text{start}}, \text{v}_{\text{next}}) \wedge \text{last}\ p = (\text{v}_0, \text{v}_1)$$

This expresses that the loop constructs a restricted walk with certain first and last edges. During construction, this is not yet a restricted path, as the last vertex is not an element of V Proving that this invariant holds is straightforward. More interesting are the cases arising after the loop terminates. The computation of the restricted path can terminate in three different ways:

- exceptionally, if no extending edge is found (due to the **RAISE** in the loop),
- exceptionally, if the constructed walk leads to the start vertex (due to the **RAISE** outside of the loop),
- or normally.

In all three cases, the function has constructed some restricted walk starting with the edge $(\text{v}_{\text{start}}, \text{v}_{\text{next}})$. In the first two cases, one needs to prove that this restricted walk excludes the existence of a restricted path starting with said edge. In the third case, the computed walk must be shown to be a restricted path. Moreover, we need to prove that the function will terminate.

In the first case, the constructed walk ends with a vertex of degree 1. Let V be the set of vertices of degree other than 2. Then this walk satisfies the preconditions of Lemma 5.7. The same holds for any restricted path w.r.t. to $V_{3,G}$ (which obviously cannot end in a vertex of degree 1. Hence this lemma can be applied and there is no restricted path (w.r.t. $V_{3,G}$) starting with $(\text{v}_{\text{start}}, \text{v}_{\text{next}})$. In a similar manner, for the second case, the restricted walk constructed by the function is the only candidate for a restricted path starting with $(\text{v}_{\text{start}}, \text{v}_{\text{next}})$. However, a restricted path cannot be a cycle, so again, no suitable restricted path exists.

For the third case and for termination, consider the following lemma. This gives an upper limit to the length of a restricted walk and guarantees that a restricted walk is a restricted path if some simple conditions on the end vertices hold.

Lemma 5.13. *Let G be a loop-free graph and V be a set of vertices with $\mathsf{V}_{3,G} \subseteq V \subseteq \mathsf{V}_G$. Moreover, let* $\mathsf{apath}_G\ V\ a\ p\ b$ *with $a \in V$. Then p and* $\mathsf{inner\text{-}verts}_G\ p$ *are both distinct lists.*

Proof. As a first step, we prove that p is not *repeating*. A walk p is repeating, if there are u, v, such that either (u, v) occurs twice in p or both (u, v) and (v, u) occur in p (in particular (u, u) may not occur in p). We prove this by induction on p.

The case $p = []$ is trivial. The same holds for $p = [(u, v)]$, as G is loop-free. Consider the case $p = as +\!\!+ [(v, w), (w, x)]$. From the induction hypothesis we know that $p' = as +\!\!+ [(v, w)]$ is not repeating. As p is a restricted walk and hence progressing, $v \neq x$ holds.

If p is repeating, then one of the following four cases applies:

1. $p' = p_0 +\!\!+ [(x', w), (w, x)] +\!\!+ p_1$ for some x', p_0, p_1. As p' is progressing, we have $x \neq x'$. Then $x' = v$ holds, as w has degree 2 (w is an inner vertex of p). But then (v, w) occurs twice in p', so p' is not repeating. This contradicts the induction hypothesis.

2. $p' = (w, x) :: p_0$ for some p_0. Then $w = a$ and hence $w \in V$. This is a contradiction to w being an inner vertex of p (and hence $w \notin V$).

3. $p' = p_0 +\!\!+ [(x, w), (w, x')] +\!\!+ p_1$ for some x', p_0, p_1. Similar to case 1, $x \neq x'$ and hence $x' = v$. Then p' contains both (w, v) and (v, w) and is therefore repeating. This contradicts the induction hypothesis.

4. $p' = p_0 +\!\!+ [(x, w)]$ for some p_0. Then $x = v$ and therefore p is not progressing. This is a contradiction to p being a restricted walk.

As all these cases can be excluded, p is not repeating and hence a distinct list.

From that one can easily derive that the inner vertices of p are distinct by contradiction: assume that the inner vertices of p are not distinct. Then there exist a vertex w and walks q_0, q_1, q_2 such that

$$p = q_0 +\!\!+ q_1 +\!\!+ q_2 \qquad \mathsf{awalk}_G\ u\ q_0\ w \qquad \mathsf{awalk}_G\ w\ q_1\ w \qquad \mathsf{awalk}_G\ w\ q_2\ v$$

and $q_0, q_1, q_2 \neq []$. Then q_0 ends with (x_0, w), q_1 with (x_1, w), and q_2 starts with (w, x_2) for some vertices x_0, x_1, x_2. As w is an inner vertex of p, it has degree 2 and therefore x_0, x_1, x_2 are not distinct. Hence, p is repeating, which is a contradiction to the assumptions. □

This leads to two important corollaries: as p is a distinct list of arcs, $|p|$ is bounded by $|\mathsf{A}_G|$. The length of p increases in every iteration of the outer loop, so the function terminates. Moreover, if, in addition to the assumptions of Lemma 5.13, the properties $v \in V$ and $u \neq v$ hold, then $\mathsf{aw\text{-}verts}_G\ u\ p = [u] +\!\!+ \mathsf{inner\text{-}verts}_G\ p +\!\!+ [v]$ is distinct and hence p is a restricted path.

From these arguments we can conclude that find-endpoint satisfies its specification. Overall, the checker satisfies the following specification:

$\forall\sigma.\ \Gamma \vdash_t \{\!|\sigma.\ \text{wf-digraph (mk-graph G)}|\!\}$
  R := **PROC** simpl_certify(G, H)
$\{\!|R \longleftrightarrow \text{certify (mk-graph G}^\sigma\text{) (mk-graph H}^\sigma\text{)} \wedge \text{loop-free (mk-graph C}^\sigma\text{)}|\!\}$

As the checker terminates, this guarantees checker correctness.

5.4. Implementation in C

The C implementation follows the same structure as the Simpl implementation. The main difference are the datatypes used: In Simpl, the abstract concepts of natural numbers, pairs and lists were used. The C implementation uses finite machine words, records and arrays instead, often stored on the heap. Finally, in Simpl, basic graph operations like "v is a vertex of G" were stated as Isabelle expressions. In C, they need to be implemented and verified.

AutoCorres translates C code into a monadic expression in Isabelle and abstracts some of the technicalities of C. The hope was that for the verification of the C program, one could start with the Simpl proof and fill in the gaps, i.e., abstract memory accesses and datatypes to the ones used in the Simpl proof and verify the functions not implemented before. The latter was indeed straightforward. Similarly, abstracting the heap to the graph datatypes of Isabelle was a bit tedious, but followed established schemes (see for example Mehta and Nipkow [53]). Most of the additional effort was needed to deal with machine words instead of natural numbers. This was somewhat surprising, because the only arithmetic operations occurring in the program are equality and increment against a fixed upper bound.

There are mainly two reasons for the problems we encountered with words: first, Isabelle has only weak support for proving properties involving words automatically. Second, such properties often occur not on their own, but as side-conditions in a larger proof. While Isabelle's automatic proof tools can often discharge such properties for natural numbers, they cannot do so for words and therefore fail, leaving the user to solve the goal mostly manually.

5.4.1. Abstraction

The issues with reasoning about words motivated me to implement an abstraction framework for AutoCorres programs. The idea is to take the original program f and transform it into a modified program f' that uses natural numbers instead of words, but otherwise has the same semantics. With the help of the abstraction framework, the two programs are proven to be equivalent. Verification of the abstracted program then implies correctness of the original program.

Abstraction or *refinement* is a well-known idea going back to Dijkstra [19] and Wirth [82] and put into a formal calculus by Back [5]. In particular, AutoCorres uses this technique to transform the Simpl program generated by the C parser into the simplified version presented to the user.

The framework presented here aims at two things: the abstraction of machine words to natural numbers and boolean values and the introduction of ghost code. Code which is added to the program purely for purposes of verification, without changing the observable behavior if the program, is traditionally called ghost code. Such ghost code is often useful for eliminating

existential quantifiers in loop invariants. For example, the function find-endpoint in Section 5.3 implicitly computes a walk in the graph. By adding ghost code, this walk can be computed explicitly in a ghost variable. This relieves the user from explicitly giving witnesses for existential quantifiers.

The abstraction framework used by AutoCorres [81] is unsuited for the addition of ghost code: it expects that each state in the concrete program corresponds to at most one state in the abstract program.

Depending on the constructs used in a C program, AutoCorres transforms it to one of several monadic languages. I defined the abstraction framework for all these languages. As these languages are very similar, I present only one variant, for a non-deterministic language with state and failure. A program in this monad has the type $(\sigma, \alpha)\ \text{prog} = \sigma \to (\alpha \times \sigma)\ \text{set} \times \text{bool}$. The first component, (α, σ) set, denotes the non-deterministic program states and the boolean flag indicates failure. Here σ is the type of the program heap and α the result type.

Definition 5.14 (Sequential Composition and Embedding). *Pure values can be embedded into this language using the operation*

$$\begin{aligned}&\text{return} : \alpha \to (\sigma, \alpha)\ \text{prog}\\ &\text{return}\ x\ s = ((x, s), \text{True})\end{aligned}$$

The sequential composition of programs is given by the operation

$$\text{bind} :\ (\sigma, \alpha)\ \text{prog} \to (\alpha \to (\sigma, \beta)\ \text{prog}) \to (\sigma, \beta)\ \text{prog}$$

which applies the second argument to all outcomes of the first program and fails if any of these computations fails.

For the values of this language, I define the define the following refinement relation.

Definition 5.15 (Refinement Relation). *Let* $a : (\alpha, \sigma)\ \text{set} \times \text{bool}$ *and* $c : (\beta, \sigma)\ \text{set} \times \text{bool}$. *Then* a refines c *w.r.t. a relation* $R \subseteq (\alpha \times \sigma) \times (\beta \times \sigma)$ *if*

$$\text{refines}\ R\ a\ c \coloneqq \neg\text{fail}\ a \implies (\neg\text{fail}\ c \wedge (\forall x_c \in \text{res}\ c.\ \exists x_a \in \text{res}\ a.\ (x_a, x_c) \in R)).$$

Here, res *refers to the set of program states and* fail *to the failure flag. We call* a *the* abstract *and the* c *the* concrete *value.*

In other words, each computation in the concrete program corresponds to a computation of the abstract program. That is, a property of the concrete program can be proved by proving this property for the abstract program, as shown in the following Lemma.

Lemma 5.16. *Let* $a : (\sigma, \alpha)\ \text{prog}$ *and* $c : (\sigma, \beta)\ \text{prog}$ *and* $R : ((\alpha \times \sigma) \times (\beta \times \sigma))\ \text{set}$. *The Hoare triple*

$$\{\!|P|\!\}\ c\ \{\!|Q|\!\}!$$

states that if the precondition $P : \sigma \to \text{bool}$ *holds before* c *is executed, then the postcondition* $P : \beta \to \sigma \to \text{bool}$ *holds after the execution and* c *did not fail. Assume that*

$$\forall h.\ P\ h \implies \text{refines}\ R\ (a\ h)\ (c\ h)$$

i.e., for all heaps satisfying the precondition P, the results of the abstract and concrete programs are related. In addition, assume that a satisfies the specification

$$\{\!|P|\!\}\ a\ \{\!|\lambda r_a, h_a.\ \forall r_c, h_c.\ ((r_a, h_a), (r_c, h_c)) \in R \implies Q\ r_c\ h_c\ |\!\}!$$

i.e., for a heap h satisfying P, the result of a h is related to a concrete program state satisfying Q. Then the concrete program c satisfies specification above.

To prove that one program refines another, I proved a set of syntax directed rules, which deconstructs two programs instruction by instruction. The rules for bind and return demonstrate the general structure of such rules.

Lemma 5.17 (Refinement of Sequential Composition). *Let* $a_1, a_2 : (\sigma, \alpha)$ prog *and* $c_1, c_2 : (\sigma, \beta)$ prog. *Then the following rule holds:*

$$\frac{\mathsf{refines}\ R\ (a_1\ h_a)\ (c_1\ h_c) \quad \forall((r_a, h_a), (r_c, h_c)) \in R.\ \mathsf{refines}\ R'\ (a_2\ r_a\ h_a)\ (c_2\ r_c\ h_c)}{\mathsf{refines}\ R'\ (\mathsf{bind}\ a_1\ a_2\ h_a)\ (\mathsf{bind}\ c_1\ c_2\ h_c)}$$

Lemma 5.18 (Refinement of Return). *Let* $a : (\sigma, \alpha)$ prog *and* $c : (\sigma, \beta)$ prog. *Then the following rule holds:*

$$\frac{((x_a, h_a), (x_c, h_c)) \in R}{\mathsf{refines}\ R\ (\mathsf{return}\ x_a\ h_a)\ (\mathsf{return}\ x_c\ h_c)}$$

Note that the relation R' in Lemma 5.17 is not fixed a priori. I use a simple type-directed solver to synthesize the relation. The basic building blocks for the relation are

- a refinement relation for natural numbers: $\{(n, w) \mid n = \mathsf{unat}\ w\}$ (here, unat is the conversion from words to natural numbers),
- a refinement relation for booleans: $\{b, w) \mid b \iff w \neq 0\}$,
- a relation for adding ghost code to a value: $\mathsf{ghost}\ R := \{(g, a)_\mathrm{G}, c) \mid (a, c) \in R\}$,
- and, as a fallback, the identity relation.

For the ghost relation, $(\cdot, \cdot)_\mathrm{G}$ is a constructor of a copy of the pair type. These building blocks are combined with a combinator for pairing relations:

$$\mathsf{pair}\ R_1\ R_2 := \{((x_a, y_A), (x_c, y_c)) \mid (x_a, x_c) \in R_1 \wedge (y_a, y_c) \in R_2\}$$

Having a separate pair type for ghost state avoids the ambiguity whether to apply the ghost or the pairing rule.

In the set of syntax directed rules, there is usually one rule per command of the language, matching this command in both languages. There are a few exceptions, where the two programs do not have exactly the same commands: adding additional commands to the abstract program, adding ghost code, and replacing procedure calls by specifications.

In many cases, a concrete operation can only be abstracted if certain preconditions are satisfied. For example, an addition on words can be replaced by an addition on natural numbers only if no overflow occurs. This requirement can be expressed by additional guard commands in the abstract program.

Definition 5.19 (Guards). *The command* guard : $(\sigma \to \text{bool}) \to (\sigma, \text{unit})$ prog *fails if the heap does not satisfy the predicate and returns a unit values otherwise.*

The guard command can also occur in the concrete program, so for adding an additional guard to the abstract program the syntactic copy $\text{guard}_{\text{add}}$ is used. A special rule combines such a guard with the following command and allows the user to use the predicate to discharge eventual side conditions.

Lemma 5.20 (Additional Guards in Refinement). *Let* $a : (\sigma, \alpha)$ prog *and* $c : (\sigma, \beta)$ prog. *Then the following rule holds:*

$$\frac{P\ h_a \implies \text{refines}\ R\ (a\ h_a)\ (c\ h_c)}{\text{refines}\ R\ (\text{bind}\ (\text{guard}_{\text{add}}\ P)\ (\lambda\_.\ a)\ h_a)(c\ h_c)}$$

See Figure 5.7 for an example. A similar situation occurs for ghost code. As ghost code may not modify the heap, it usually suffices to insert return commands. Again, those commands are marked with $_{\text{add}}$.

Lemma 5.21 (Adding Ghost Code). *Let* $a : (\sigma, \alpha)$ prog *and* $c : (\sigma, \beta)$ prog. *Then the following rules hold:*

$$\frac{\text{refines}\ R\ (a\ x\ h_a)\ (c\ h_c)}{\text{refines}\ R\ (\text{bind}\ (\text{return}_{\text{add}}\ x)\ a\ h_a)\ (c\ h_c)}$$

$$\frac{\text{refines}\ R\ (\text{bind}\ a\ (\lambda\_.\ \text{return}\ x)\ h_a)\ (\text{bind}\ c\ \text{return}\ h_c)}{\text{refines}\ R\ (\text{bind}\ a\ (\lambda\_.\ \text{return}_{\text{add}}\ x)\ h_a)\ (c\ h_c)}$$

The first rule allows inserting ghost code in the middle of the program, the second rule appending it to the end.

As a last point, AutoCorres programs can contain user-defined procedures. During refinement, one can replace a procedure call by a specification of the procedure. If this procedure corresponds to a pure function, it can even be replaced by a return.

Lemma 5.22.

$$\frac{\forall P.\ \{\!|\lambda h_c.\ P\ (f\ h_c) \wedge Q\ h_c|\!\}\ c\ \{\!|P|\!\}!\quad Q\ h_c\quad ((v, h_a), (f\ h_c, h_c)) \in R}{\text{refines}\ R\ (\text{return}\ v\ h_a)\ (c\ h_c)}$$

Here, the first premise is a specification for a procedure computing a pure function f in the format typically used in AutoCorres.

Putting abstraction to use For the Kuratowski checker, the proof process is as follows: for each function f containing word arithmetic, I make a copy f' of this function, in which words are replaced by natural numbers and booleans. For each arithmetic operation, a guard is inserted, stating that this operation would not overflow on words (see Figure 5.7). Where necessary, I also add ghost code and annotate loops with invariants. One example of this is function find-endpoint (see Figure 5.5), where I add a variable holding the computed restricted path and use this in the invariant. Note that the ghost code can use arbitrary Isabelle expressions. Then I prove that f' is an abstraction of f, using the verification condition generator sketched in the previous section. The proof is mostly automatic; only simple properties about words and their relation to natural numbers and booleans need to proven.

	guard (λ_. ($i : \mathbb{N}$) < word-max32);
return ((i : word32) + 1)	return (i + 1)
(a) concrete program	(b) abstract program

Figure 5.7.: Abstraction of word arithmetic.

5.5. Conclusion

In this chapter, I formalized a class of certificates for the non-planarity of graphs and verified two checkers of such certificates. Following the implementation in the LEDA library, the certificates are not necessarily Kuratowski subgraphs, but certain additional features are allowed. This allows for a simpler implementation of the checkers, as these do not need to check whether a graph is connected or free of parallel paths.

I implemented the checker both in an abstract language, Simpl, and in C and discussed the differences in the verification. Interestingly, the bulk of additional effort needed for the verification of the C program was not due to memory handling, but due to the use of word arithmetic instead of natural numbers. The reason was that words permeated the whole proof, which made automatic proof tools much less effective. This was solved by a developing an abstraction mechanism, which also added the ability to use ghost code.

After abstraction, verification of the C implementation followed closely the proof of the Simpl program. Therefore, AutoCorres provides a viable option for the verification of C programs and the verification can profit from a previous verification of pseudo code, for example the Simpl program. However, the C program must be lifted to a similar level of abstraction first.

Since I did this formalization, AutoCorres gained the ability to automatically abstract word arithmetic to natural numbers. This works well on the C program presented here. Had this been available earlier, there would have been no incentive to develop a new abstraction framework. However, my abstraction framework provides the additional option of inserting ghost code and can be used on top of the automatic abstraction used by AutoCorres.

The characterization of the certificates accepted by the certify procedure consists of around 2000 lines of proof. Around a quarter of explicitly deals with contracting and subdividing walks and would have been more convenient to prove in a formalization supporting undirected graphs natively. Both the Simpl and the C implementation of the Kuratowski checker consist of around 300 lines of code (the Simpl syntax is more verbose than C). The verification of the Simpl checker was done in 1300 lines. The verification of the C checker required 3200 lines and 1400 lines for the refinement framework. Of the 3200 lines, 900 deal with heap abstraction, heap access, and the verification of basic graph operations not implemented in the Simpl code.

The verified Simpl implementation is available in the Archive of Formal Proofs [64]. The results for the C implementation including the full set of rules for the abstraction framework can be found on the homepage of the author[3].

[3] http://www21.in.tum.de/~noschinl/Non_Planarity_Certificate/.

6. Structured Proofs in Program Verification

Verification of (imperative) programs usually follows a certain scheme: the program and its specification are transformed into a set of logic formulas, called *verification conditions* (VC). These formulas are proven valid, which implies that the program obeys its specification. An algorithm performing the transformation into verification conditions is called *verification condition generator* or short *VCG*. Even though the verification conditions are generated by a structural decomposition of the program, it is often not trivial to relate a formula to the part of the program it describes. In this chapter, I present *case labeling*, a technique to generate labeled verification conditions. Together with Isabelle's facilities for writing structured proofs, this allows the user to write proofs where it is immediately clear which part of the proof relates to which part of the program.

An early version of this work was presented by me at the Isabelle Workshop at ITP 2015 [66].

6.1. A Simple Imperative Language

To demonstrate the labeling technique, I present a simple language and a corresponding verification condition generator. The language $\mathcal{L}$ defined in this section is a simplified version of the languages used in the AutoCorres tool for the verification of C programs [33]. This language is a shallow embedding of computations with failures, i.e., $\mathcal{L}$ programs are represented as values of type α option. The value None represents failure (including non-termination), while Some x represents the successful computation of a value x. Local variables are emulated by lambda abstractions.

The language is built from four combinators: bind for sequential composition, return for embedding pure expressions, while for loops and cond for conditional statements. The definitions of these combinators are given in Figure 6.1. For longer $\mathcal{L}$-programs, I use a nicer syntax: $x \leftarrow c_1;\ c_2\ x$ for $\mathsf{bind}\, c_1\, (\lambda x.\, c_2\, x)$ and **WHILE** b **INV** I **BODY** c **START** x for $\mathsf{while}\, b\, I\, c\, x$. The whole program is then wrapped by **DO** ... **OD**. This notation is inspired by Haskell's do-notation for monads.

A specification of a program c is expressed as a Hoare triple $\{\!|P|\!\}\ c\ \{\!|Q|\!\}$, where P is the precondition, i.e., a formula which holds before c is executed and Q the postcondition, i.e., a predicate on the result of c.

Definition 6.1 (Hoare triples for $\mathcal{L}$). *The Hoare triple $\{\!|P|\!\}\ c\ \{\!|Q|\!\}$ is valid if for all r holds:*

$$P \wedge c = \mathsf{Some}\ r \implies Q\ r$$

This is a predicate for partial correctness; it holds trivially if the program does not terminate.

For the verification condition generator, I use a weakest precondition calculus. The rules of this calculus are given in Figure 6.2. Starting with a Hoare triple, the verification condition

```
definition bind : α option → (α → β option) → β option where
  bind c1 c2 ≔ (case c1 of None → None | Some x → c2 x)

definition return : α → α option where
  return x ≔ Some x

partial_function (option) while : (α → bool) → (α → bool)
    → (α → α option) → (α → α option) where
  while b I c s ≔ if b s then bind (c s) (while b I c) else s

definition cond : bool → α option → α option → α option where
  cond b c1 c2 ≔ if b then c1 else c2
```

Figure 6.1.: Commands of $\mathcal{L}$. The second parameter of while is an annotation: the VCG uses it as the loop invariant. A nonterminating loop fails, i.e., returns None.

$$\frac{}{\{\!|P\ x|\!\}\ \mathsf{return}\ x\ \{\!|P|\!\}}\ \text{Return} \qquad \frac{\forall x.\ \{\!|R\ x|\!\}\ c_2\ x\ \{\!|Q|\!\} \quad \{\!|P|\!\}\ c_1\ \{\!|R|\!\}}{\{\!|P|\!\}\ \mathsf{bind}\ c_1\ c_2\ \{\!|Q|\!\}}\ \text{Bind}$$

$$\frac{\forall x.\ \{\!|I\ x \wedge b\ x|\!\}\ c\ x\ \{\!|I|\!\} \quad \forall x.\ I\ x \wedge \neg b\ x \implies P\ x}{\{\!|I\ x|\!\}\ \mathsf{while}\ b\ I\ c\ x\ \{\!|P|\!\}}\ \text{While}$$

$$\frac{\{\!|P_1|\!\}\ c_1\ \{\!|Q|\!\} \quad \{\!|P_2|\!\}\ c_2\ \{\!|Q|\!\}}{\{\!|\mathsf{if}\ b\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2|\!\}\ \mathsf{cond}\ b\ c_1\ c_2\ \{\!|Q|\!\}}\ \text{Cond}$$

$$\frac{\{\!|P'|\!\}\ c\ \{\!|Q|\!\} \quad P' \implies P}{\{\!|P|\!\}\ c\ \{\!|Q|\!\}}\ \text{Conseq}$$

Figure 6.2.: Weakest precondition calculus for $\mathcal{L}$ for partial correctness.

generator applies the rules Return, Bind, While, and Cond whenever possible, solving the premises from left to right. If this fails because the precondition does not match, it tries applying Rule Conseq first. If this process introduces fresh variables, these start out being "schematic" and are then instantiated as necessary when applying further rules. Rule WhileT from Figure 6.3 is an alternative to Rule While, which could be used in a setup for total correctness (that is, if a valid Hoare triple requires a non-failing program).

6.2. Problem Statement

When we reason (informally) about a program, we usually have the operational semantics of the language in mind, focusing our thoughts to certain parts of the program. The goal of case labeling is to support the user by supplying the necessary information.

$$\frac{\forall x.\ \{\!|I\ x \wedge b\ x|\!\}\ c\ x\ \{\!|\lambda x'.\ I\ x' \wedge (x', x) \in R|\!\} \quad \forall x.\ I\ x \wedge \neg b\ x \implies P\ x}{\{\!|I\ x|\!\}\ \text{while}\ b\ I\ c\ x\ \{\!|P|\!\}}\ \text{WhileT}$$

Figure 6.3.: Variant of Rule While for total correctness. If R is well-founded, this rule guarantees termination.

Example 6.2 (Largest Odd Divisor). *Consider the following $\mathcal{L}$-program which computes the largest odd divisor of a positive integer a.*

$\{\!|0 < a|\!\}$
DO
  $n \leftarrow \text{return}\ \ a$;
  WHILE $(\lambda n.\ \text{even}\ n)$
  INV $(\lambda n.\ 0 < n \wedge n\ \text{dvd}\ a \wedge (\forall m.\ \text{odd}\ m \wedge m\ \text{dvd}\ a \implies m\ \text{dvd}\ n))$
  BODY $(\lambda n.\ \text{return}\ (n/2))$
  START n
OD
$\{\!|\lambda r.\ \text{odd}\ r \wedge r\ \text{dvd}\ a \wedge (\forall m.\ \text{odd}\ m\ \wedge m\ \text{dvd}\ a \implies m \leq r)|\!\}$

We convince ourselves that, before and after each iteration of the loop, n is a divisor of a and each odd divisor of a divides n. This is the loop invariant. If the loop ends, n is an odd divisor of a and any other odd divisor m of a divides n (and hence $m \leq n$). This proves the correctness of the program.

Using the rules of Figure 6.2, the verification condition generator decomposes this Hoare triple into verification conditions, i.e. pure logical formulas, not involving any statements of the language:

$$\begin{aligned}&\forall x, y.\ 0 < y \wedge y\ \text{dvd}\ a \wedge (\forall m.\ \text{odd}\ m \wedge m\ \text{dvd}\ a \implies m\ \text{dvd}\ y) \wedge \text{even}\ y\\ &\quad\implies 0 < y/2 \wedge y/2\ \text{dvd}\ a \wedge (\forall m.\ \text{odd}\ m \wedge m\ \text{dvd}\ a \implies m\ \text{dvd}\ y/2)\end{aligned} \tag{6.1}$$

$$\begin{aligned}&\forall x, y.\ 0 < y \wedge y\ \text{dvd}\ a \wedge (\forall m.\ \text{odd}\ m \wedge m\ \text{dvd}\ a \implies m\ \text{dvd}\ y) \wedge \neg\,\text{even}\ y\\ &\quad\implies \text{odd}\ y \wedge y\ \text{dvd}\ a \wedge (\forall m.\ \text{odd}\ m \wedge m\ \text{dvd}\ a \implies m < y)\end{aligned} \tag{6.2}$$

$$0 < a \implies 0 < \wedge a\ \text{dvd}\ a \wedge (\forall m.\ \text{odd}\ m \wedge m\ \text{dvd}\ a \implies m\ \text{dvd}\ a) \tag{6.3}$$

Note that there is no explicit reference to the program structure left in these verification conditions. On the other hand, the rules used by the verification condition generator are structural: the program is decomposed step by step into statements and each of these steps might generate a verification condition. So each of the verification conditions belongs to some part of the program.

By closely inspecting the formulas we can recover this information manually: the verification condition (6.1) has an instance of the invariant as assumption and as conclusion, so this condition describes that the invariant is preserved by the loop. Then, (6.2) describes that after the loop

finishes, the postcondition must hold. Finally, $0 < a$ is the precondition of the program and $n = a$ before the while loop. So (6.3) is the obligation to show that the precondition of the specification implies the weakest precondition computed by the VCG.

There are bound variables x and y in (6.1) and (6.2). These names never occurred in the program text. The y is the variable of the loop state (which was consistently called n in the program). The x was introduced by Rule Bind for the result of the initial return. As this value is not explicitly mentioned in the annotated invariant, it does not occur in the formulas otherwise.

This information is important: We want to use our intuition about the program to guide the proof. For this small example it was relatively easy to correlate verification conditions and parts of the program: we combined our knowledge of the used Hoare calculus with our informal understanding of the program and matched that against the generated verification conditions. For larger programs, this process can become very tedious. In particular, this is the case when the verification conditions are different then expected; because the program is not correct, the invariants are too weak, or our intuitive understanding is simply wrong.

This manual inference should not be necessary: the verification condition generator can keep track of its position in the program and somehow attach this information to the generated verification conditions. This idea is not new: ESC/Java shows column and line number when displaying a counter-example for a verification condition it could not prove, as well as the branching points taken to arrive at this position [46]. Similarly, JACK provides this information by displaying a highlighted version of the source code[8] and VCC and Spec# integrate into Visual Studio, providing the information directly in the editor [17].

6.2.1. Structured Proofs in Isabelle/Isar

Unlike the specialized software verification tools mentioned in the last section, Isabelle is a general purpose proof assistant. Hence its proof language is not specifically tailored to program verification. Instead it provides sophisticated generic constructs for structuring proofs; in particular the *cases* mechanism.

Example 6.3. *With an appropriate VCG – the components of which will be introduced in this chapter – the proof for the example in Section 6.2 can be structured as follows:*

```
proof vcg
 { case conseq ... }        specification precondition implies weakest precondition: (6.3)
 { case (bind _)            variable not used in the following cases, so no name necessary
  { case (while n)
    { case inv ... }        loop preserves invariant: (6.1)
    { case post ... }       invariant after loop implies postcondition: (6.2)
  }
 }
qed
```

The commands proof *and* qed *indicate a proof block. The initial proof method* vcg *generates the verification conditions and associated cases. A* **case** *selects one of these obligations, providing the user with names for the assumptions and shortcuts for the conclusion and other important terms. The syntax* **case** (while n) *gives a name to a universally quantified variable, in our example the y*

in (6.1) *and* (6.2)*. Such cases can be arbitrarily named and nested. The ... are placeholders for the actual proofs.*

Even to a casual reader, the structure of the proof is immediately obvious. This is in stark contrast to so-called unstructured proofs: a list of tactics, which are applied in sequence to the proof obligations and modify these until they vanish. Such proofs are hard to read (essentially, one needs to step through the proof, investigating the results of each tactic) and one is restricted to a small fragment of the proof language. It is possible to write structured proofs without the cases mechanism. For this, all premises and the conclusion must be explicitly mentioned in the proof text. But for program verification, these are large: a program which can be verified by a few lines of unstructured proofs easily needs more than a page just to write down the components of the proof obligations.

In the remainder of this section, I introduce the basics of Isabelle's handling of structured proofs. Isabelle manages the proof obligations for a theorem in a *goal state*. For our purposes, a goal state is a list of proof obligations, called *subgoals*.

Definition 6.4 (Subgoal)**.** *A subgoal is a logical formula φ of the form*

$$\forall x_1.\ \ldots \forall x_m.\ P_1 \Longrightarrow \ldots \Longrightarrow P_m \Longrightarrow Q. \tag{6.4}$$

Here, the P_i are the assumptions of the subgoal and concl $\varphi = Q$ *is the conclusion. We also write $\forall x_1, \ldots, x_m$ instead of $\forall x_1.\ \ldots \forall x_m$ and call the x_i are* top-level quantified.

Structured proofs in Isabelle/Isar [79] build on the concept of *proof contexts*. A proof context consists of a list of fixed variables (*fixes*), a list of named local assumptions, and a list of term bindings, i.e., syntactic abbreviations for terms. Inside a proof context, the assumptions can be used as any other theorem. The order of the fixes in the proof context does not need to coincide with the order of top-level quantified variables in the subgoal.

Proof contexts can be nested. Such a nested proof context inherits the contents of the outer proof context. When leaving a nested proof context, the additional contents vanish. A nested context is indicated with { ... } in a proof text.

The cases mechanism is a way to setup proof contexts such that the user does not need to write down the assumptions etc. explicitly. Each case has a name and consists of a list of fixes, a named lists of assumptions, a list of term bindings, and a list of nested cases. The **case** *name* command enriches the current proof context with the fixes, assumptions, and bindings of the case *name* and makes its nested cases available. The user can override the default names of the fixes by using the **case** (*name* x_1 x_2 ...) syntax.

6.3. Labeled Subgoals

In this section, I present a labeling of subgoals from which a meaningful cases structure can be generated, without looking at the semantics of a subgoal. For this annotation, I use *labeling constants*, that is, functions which can be inserted into the subgoal without changing the semantics.

An important design criterion is that the labeling must encode the cases structure in a way that proof methods following the usual conventions of the prover can easily generate. Ideally,

the labeling is just an add-on: Adding support for labeling to a proof tool should not require functional changes.

Ignoring the nested nature of cases for now, we describe name and fixes of a case using blocks.

Definition 6.5 (Block). *A* block $b = \langle x, n, ts\rangle_B$ *is a triple of a name* x*, a statement number* n*, and a list of terms* ts *representing (top-level quantified) variables. We write* $\mathsf{num}\ b = n$ *and* $\mathsf{vars}\ b = ts$ *to refer to the components of a block. Moreover,* $\mathsf{iden}\ b = (n, x)$ *is the* block identifier.

The statement number of a block will be used to determine the order relative to other blocks. By using a list of blocks we can also describe the nesting of blocks. This is called a *context*. A *suffix* of the context xs is a context zs satisfying $\exists ys.\ xs = ys +\!\!+ zs$. If ys is non-empty, zs is a proper suffix.

Definition 6.6 (Subgoal Labeling). *The labeling constant* vc *takes a non-empty list of blocks* ct *and wraps some term* t*, that is,* $\mathsf{vc}\ ct\ t \coloneqq t$.

This constant describes which case should be associated to a subgoal and is applied to the conclusion of the subgoal (and not the whole subgoal). This way, it does not interfere with Isabelle's standard methods of applying rules.

The idea is that activating this case should setup a proof context which is suitable to prove this subgoal. I.e., if a subgoal φ has the form

$$\forall x_1.\ \ldots \forall x_m.\ P_1 \Longrightarrow \ldots \Longrightarrow P_m \Longrightarrow Q.$$

and the user activates the case associated with this subgoal, the proof context should fix variables $x_1, \ldots, x_n$, assume $P_1, \ldots, P_m$, and contain a term binding for Q. The associated case and its ancestors are described by ct, where the first element is the innermost case.

For reasons of readability, I sometimes place the labeling constants above or below the term they wrap, indicating the scope by a brace.

Example 6.7. *Recall Example 6.2. The verification conditions can be labeled as follows:*

$$\begin{aligned}&\forall x, y.\ 0 < y \wedge y \ \mathsf{dvd}\ a \wedge (\forall m.\ \mathsf{odd}\ m \wedge m \ \mathsf{dvd}\ a \Longrightarrow m \ \mathsf{dvd}\ y) \wedge \mathsf{even}\ y \\ &\Longrightarrow \overbrace{0 < y/2 \wedge y/2 \ \mathsf{dvd}\ a \wedge (\forall m.\ \mathsf{odd}\ m \wedge m \ \mathsf{dvd}\ a \Longrightarrow m \ \mathsf{dvd}\ y/2)}^{\mathsf{vc}\ [\langle 4, \text{«conseq»}, []\rangle_B, \langle 3, \text{«inv»}, []\rangle_B, \langle 3, \text{«while»}, [y]\rangle_B, \langle 2, \text{«bind»}, [x]\rangle_B]}\end{aligned} \tag{6.5}$$

$$\begin{aligned}&\forall x, y.\ 0 < y \wedge y \ \mathsf{dvd}\ a \wedge (\forall m.\ \mathsf{odd}\ m \wedge m \ \mathsf{dvd}\ a \Longrightarrow m \ \mathsf{dvd}\ y) \wedge \neg\, \mathsf{even}\ y \\ &\Longrightarrow \overbrace{\mathsf{odd}\ y \wedge y \ \mathsf{dvd}\ a \wedge (\forall m.\ \mathsf{odd}\ m \wedge m \ \mathsf{dvd}\ a \Longrightarrow m < y)}^{\mathsf{vc}\ [\langle 6, \text{«post»}, []\rangle_B, \langle 3, \text{«while»}, [y]\rangle_B, \langle 2, \text{«bind»}, [x]\rangle_B]}\end{aligned} \tag{6.6}$$

$$0 < a \Longrightarrow \overbrace{0 < a \wedge a \ \mathsf{dvd}\ a \wedge (\forall m.\ \mathsf{odd}\ m \wedge m \ \mathsf{dvd}\ a \Longrightarrow m \ \mathsf{dvd}\ a)}^{\mathsf{vc}\ [\langle 0, \text{«conseq»}, []\rangle_B]} \tag{6.7}$$

Both (6.5) and (6.6) arise from Rule Conseq*. The former arises from the beginning of the loop body of the while loop, the latter at the beginning of the program. The verification condition (6.6) arises from the same while loop as (6.5). In both,* y *refers to the loop variable.*

The contexts of these three proof obligations are

(6.5): $[\langle 4, \text{«conseq»}, [] \rangle_B, \langle 3, \text{«inv»}, [] \rangle_B, \langle 3, \text{«while»}, [y] \rangle_B, \langle 2, \text{«bind»}, [x] \rangle_B]$

(6.6): $[\langle 6, \text{«post»}, [] \rangle_B, \langle 3, \text{«while»}, [y] \rangle_B, \langle 2, \text{«bind»}, [x] \rangle_B]$

(6.7): $[\langle 0, \text{«conseq»}, [] \rangle_B)]$

The absolute values of the statement numbers do not matter at the moment. The context of the first two have the common suffix $[\langle 3, \text{«while»}, [y] \rangle_B, \langle 2, \text{«bind»}, [x] \rangle_B]$*, so both proof obligations can be grouped together. Comparing the statement numbes with a lexicographic ordering yields* $[0] < [2, 3, 3, 4] < [2, 3, 6]$ *and hence the order* (6.7), (6.5), (6.6)*. Overall, we get the structure seen in Example 6.3.*

With the labeling presented so far, we can attach names to proof obligations and order them. This already turns a list of subgoals into a tree reflecting the program structure. The next labeling constant allows to add more structure to a single proof obligation.

As already mentioned, the conclusion will be implicitly used as a term binding. Additional term bindings can also be declared explicitly.

Definition 6.8 (Term Binding Labeling). *The labeling constant* tbind *takes a name* x*, a statement number* n*, and wraps a term* t*, that is,* $\text{tbind}\ x\ n\ t := t$.

Again, the number will be used for sorting purposes. The tbind label has two functions. First, it gives a name to the wrapped term, that is, if a proof obligation contains the subterm $\text{tbind}\ x\ n\ t$, a term binding x with term t is added. Second, if tbind is applied to the conclusion of an assumption, it gives a name to the assumption.

Example 6.9. *Recall Example 6.2, but using Rule* WhileT *with the relation* $<$ *for* R *instead of Rule* While*. Then, instead of* (6.1)*, we get*

$$\begin{aligned} &\forall x, y.\ 0 < y \land y \text{ dvd } a \land (\forall m.\ \text{odd}\ m \land m \text{ dvd } a \implies m \text{ dvd } y) \land \text{even}\ y \\ &\quad\implies 0 < y/2 \land y/2 \text{ dvd } a \land (\forall m.\ \text{odd}\ m \land m \text{ dvd } a \implies m \text{ dvd } y/2) \land y/2 < y \end{aligned} \tag{6.8}$$

which can be labeled as

$$\begin{aligned} &\forall x, y.\ 0 < y \land y \text{ dvd } a \land (\forall m.\ \text{odd}\ m \land m \text{ dvd } a \implies m \text{ dvd } y) \land \text{even}\ y \\ &\implies \overbrace{\underbrace{0 < y/2 \land y/2 \text{ dvd } a \land (\forall m.\ \text{odd}\ m \land m \text{ dvd } a \implies m \text{ dvd } y/2)}_{\text{tbind «inv» 3}} \land \underbrace{y/2 < y}_{\text{tbind «var» 3}}}^{\text{vc } [\langle 4, \text{«conseq»}, [] \rangle_B, \langle 3, \text{«inv»}, [] \rangle_B, \langle 3, \text{«while»}, [y] \rangle_B, \langle 2, \text{«bind»}, [x] \rangle_B]} \end{aligned} \tag{6.9}$$

Due to the two tbind *labels, the following term bindings*

$$\begin{aligned} &\text{«inv»} := 0 < y/2 \land y/2 \text{ dvd } a \land (\forall m.\ \text{odd}\ m \land m \text{ dvd } a \implies m \text{ dvd } y/2 \\ &\text{«var»} := y/2 < y \end{aligned}$$

will be added to the case «conseq»*. The first corresponds to the invariant, the second to the variant, that is, the termination condition.*

Naming of assumptions will be demonstrated in the next example. By default, all assumptions will be associated with the case of the innermost context block. For example, in Example 6.9, the single assumption is associated to the case «conseq», that is this assumption is only available after this case (and all the cases above) are activated. Sometimes, it is useful to associate an assumption with one of the cases above instead. This can be done with the following labeling constant.

Definition 6.10 (Hierarchy Labeling). *The labeling constant* hier *takes a context ct and wraps a term t, that is,* hier *ct t* ≔ *t*

This constant can be applied to the conclusion of an assumption. The context argument *ct* describes to which case of the context of the subgoal this assumption is associated.

Example 6.11. *Consider the following two proof obligations.*

$$\text{hier}\ [(\text{«then»}, 0, [])]\ (\text{tbind «cond»}\ 0\ b) \implies \text{vc}\ [(\text{«cond1»}, 1, []), (\text{«then»}, 0, [])]\ c_1 \quad (6.10)$$

$$\text{hier}\ [(\text{«then»}, 0, [])]\ (\text{tbind «cond»}\ 0\ b) \implies \text{vc}\ [(\text{«cond1»}, 1, []), (\text{«then»}, 0, [])]\ c_1 \quad (6.11)$$

Here, the hier *labeling identifies the assumption b in both obligations and associates it with the case* «then»*, which is in the common suffix of the contexts of these two obligations. The* tbind *labeling gives the name* cond *to this assumption. This allows for the following proof structure:*

```
{ case then                    from here on cond refers to the theorem b
  { case cond1 ... }
  { case cond2 ... }
}
```

That is, after the case «then» *has been activated, the theorem b is available under the name* «cond»*. Without the* hier *and* tbind *labels, the theorem b would only be available in the cases* «cond1» *and* «cond2»*, under some default name.*

The last constant I introduce is not part of the labeling per se, but is used to compute the labeling. It will be discussed in detail later in this section.

Definition 6.12 (Context Helper). *The labeling constant* ctxt *takes a statement number, a context, another statement number, and wraps a term t, that is,* ctxt *i ct o t* ≔ *t*.

To express the parameters of the labeling constants in the logic, we need a way of embedding tuples, natural numbers, strings, and a list of terms into the logic. This is not a restriction in practice. For the remainder of this chapter, we assume such an embedding and silently convert from and to it where necessary.

For a single subgoal it is always possible to generate an appropriate cases structure from such a labeling. For a set of subgoals however, we need to make sure that the descriptions of the cases are not contradictory: i.e., the same case described in different subgoals must have the same fixes and assumptions and the term bindings should be either the same or have different names.

We have to be careful about what "the same fixes" means: top-level quantified variables are local to a single subgoal, so what does it mean if two top-level quantified variables are "the same" in two subgoals? I define this notion by referring to the variables recorded in the contexts.

Definition 6.13 (Wellformed Variable Recording). *We say a subgoal φ is* labeled*, if* concl φ *has the form* vc *ct P or* ctxt *i ct o P for some i, o, ct, and P. For such a verification condition, we write* ct_φ *for ct. For a context ct, let* varss *ct* = flat (map vars *ct*) *be the concatenation of the term lists of a context.*

Then, ct_φ *is a* wellformed variable recording *of φ, if* varss *ct is distinct, i.e., does not contain any term twice, and contains only top-level quantified variables of φ.*

Definition 6.14 (Equality of Top-level Quantified Variables). *Let φ_1, φ_2 be two subgoals with a wellformed variable recording and let ct_1 resp. ct_2 be the longest suffixes of* ct_{φ_1} *resp.* ct_{φ_2} *such that*

$$\mathsf{map}\ \mathsf{iden}\ ct_1 = \mathsf{map}\ \mathsf{iden}\ ct_2.$$

We say ct_{φ_1} *and* ct_{φ_2} *are* compatible contexts*, if the variables in ct_1 and ct_2 have the same type, i.e.*

$$\mathsf{map}\ \mathsf{type}\ (\mathsf{varss}\ ct_1) = \mathsf{map}\ \mathsf{type}\ (\mathsf{varss}\ ct_2)$$

Then, top-level quantified variables of φ_1 resp. φ_2 are equal if they occur at the same position in varss ct_1 *resp.* varss ct_2.

This means that two variables can be compared only if they occur in the common suffix of the context. Note that this definition can easily be lifted to compare arbitrary subterms of a subgoal.

Example 6.15. *Consider the following two subgoals.*

$$\forall x_1 : \alpha, x_2 : \gamma, x_3 : \gamma, x_4 : \gamma.\ \mathsf{vc}\ [(\text{«c»}, 2, [x_1]), (\text{«b»}, 1, [x_3]), (\text{«a»}, 0, [x_2, x_4])]\ P \tag{6.12}$$

$$\forall y_1 : \alpha, y_2 : \beta, y_3 : \gamma, y_4 : \gamma.\ \mathsf{vc}\ [(\text{«d»}, 3, [y_1]), (\text{«b»}, 1, [y_2]), (\text{«a»}, 0, [y_3, x_4])]\ Q \tag{6.13}$$

The longest "common" context suffixes are

$$ct_1 = [(\text{«b»}, 1, [x_3]), (\text{«a»}, 0, [x_2, x_4])]$$
$$ct_2 = [(\text{«b»}, 1, [y_2]), (\text{«a»}, 0, [y_3, y_4])]$$

which yields varss $ct_1 = [x_3, x_2, x_4]$ *and* varss $ct_2 = [y_2, y_3, y_4]$*. Then x_2 and y_3 as well as x_4 and y_4 are equal, but x_2 and y_4 respectively x_4 and y_3 are not equal (because the occur at different positions). The variables x_3 and y_2 cannot be compared because they have different types (hence, the contexts ct_1 and ct_2 are not compatible) and the variables x_1 and y_1 cannot be compared because they do not occur in the common suffix.*

Definition 6.16 (Wellformed Labeling). *The* hierarchy context *of a term t in a subgoal φ is*

$$\mathsf{hct}_\varphi\ t = \begin{cases} ct & \text{if } t = \mathsf{hier}\ ct\ t' \\ \mathsf{ct}_\varphi & \text{else} \end{cases}$$

Then, a set S of subgoals has a wellformed labeling *if the following conditions are fulfilled:*

1. *Each verification condition $\varphi \in S$ is labeled and has a wellformed variable recording,*

2. *the hierarchy contexts are valid, i.e.,* $\mathsf{hct}_\varphi\ t$ *is a suffix of* ct_φ *for all subterms* t *of* φ*,*
3. *if a variable is free in a term binding in* φ*, then it is either top-level quantified or free in* φ*,*
4. *if* φ *is labeled with* vc*, then the first block of* ct_φ *does not occur in* ct_{φ_2} *for any other* $\varphi_2 \in S$*,*
5. *the assumptions are consistent, i.e., an assumption* P *of* φ *is also an assumption of any* $\varphi_2 \in S$ *where* $\mathsf{hct}_\varphi\ P$ *is a suffix of* ct_{φ_2}*, and*
6. *all contexts are compatible, i.e., for all* $\varphi_1, \varphi_2 \in S$ *holds* ct_{φ_1} *and* ct_{φ_2} *are compatible.*

Such a labeling is called final*, if each verification condition is labeled with* vc*.*

Condition 2 stems directly from Definition 6.10. Term bindings must make sense outside of the subterm they are defined in, hence condition 3. Due to condition 4, each subgoal is associated to a unique leaf case. The conditions 5 and 6 arise as we want to group verification conditions with the same context-suffix (as in Example 6.7): this is only possible if these blocks fix the same variables and introduce the same assumptions[1].

In the remainder of this section, I describe how to systematically label introduction rules to produce a wellformed labeling. Porting this technique to other methods of computing verification conditions (e.g., equation-based as in [32]) should be straightforward.

We have already seen the labeling constant ctxt. Similar to the subgoal labeling vc, it attaches to the conclusion of a subgoal. This constant is used to compute statement numbers and contexts and will not end up in the final verification conditions. The statement number is used to obtain an ordering of the cases and to distinguish unrelated blocks with the same name. For program verification, the statement number would very roughly correspond to a line number in the program. In ctxt i ct o P, i is the statement number of the program fragment in P, ct the context, and o is the first statement number after the program fragment in P. Initially, we fix $i = 0$, $ct = []$ and use Isabelle's schematic variables to compute o. This is expressed by the following rule:

$$\frac{\mathsf{ctxt}\ 0\ []\ o\ P}{P}\ \mathrm{Label_C}$$

In Definition 6.16, we have seen a wellformedness condition for a set of subgoals. For computing a labeled set of subgoals, we need to know which kind of transformations preserve the wellformedness. Below, I describe a set of local conditions for introduction rules, which are sufficient to guarantee that the application of such a rule obeys the global requirements of Definition 6.16.

A basic requirement is that a rule can introduce fresh blocks without colliding with other blocks in one of the subgoals. To achieve this, the labeling ctxt i ct o P reserves the statement numbers in the interval $\{n \mid i \leq n < o\}$ for subgoals derived from P, i.e., those numbers may not occur anywhere else in the goal state. A set of subgoals satisfying this condition has a position-wellformed labeling.

[1] Due to α-equivalence of lambda terms, the names of variables do not matter.

Definition 6.17 (Position-Wellformed Labeling). *The sets of* reserved numbers, blocks, *and* new numbers *are defined as follows:*

$$\text{res-num}\,\varphi = \begin{cases} \{n \mid i \le n \le o\} & \text{if } \text{concl}\,\varphi = \text{ctxt}\; i\; ct\; o\; t' \\ \varnothing & \text{else} \end{cases}$$

$$\text{blocks}\,\varphi = \text{set}\;\; \text{ct}_\varphi$$

A set of subgoals S has a position-wellformed labeling if it has a wellformed labeling and for all $\varphi_1, \varphi_2 \in S$ holds

$$\text{res-num}\,\varphi_1 \cap \text{num}\;(\text{blocks}\,\varphi_2) = \varnothing$$
$$\varphi_1 \neq \varphi_2 \implies \text{res-num}\,\varphi_1 \cap \text{res-num}\,\varphi_2 = \varnothing.$$

Definition 6.18 (Wellformed Labeled Rule). *Let* $\text{new-num}\,P_i = \text{res-num}\,P_i \cup \text{num}\;(\text{blocks}\,P_i \setminus \text{blocks}\,Q)$ *and consider an inference rule*

$$\frac{P_1 \quad \dots \quad P_n}{Q}. \tag{6.14}$$

This is a wellformed labeled rule *if all of the following conditions hold for any $1 \le i, j \le n$:*

1. *P_i is labeled,*
2. *ct_Q is a suffix of ct_{P_i},*
3. *new numbers are reserved by Q, i.e.,* $\text{new-num}\,P_i \subseteq \text{res-num}\,Q$,
4. *all newly recorded variables are recorded at most once and are top-level quantified in P_i, i.e.,* $\text{vars}\;(\text{blocks}\,P_i \setminus \text{blocks}\,Q)$ *are top-level quantified variables in P_i,*
5. *for any premise P in P_i, $\text{hct}_{P_i}\,P$ is a suffix of ct_{P_i} and has ct_Q as a proper suffix,*
6. *any premise P of P_i is also an premise of P_j, if ct_{P_j} is a suffix of $\text{hct}_{P_i}\,P$,*
7. *reserved numbers occur at most once, i.e.,* $\text{res-num}\,P_i \cap \text{num}\;(\text{blocks}\,P_j \setminus \text{blocks}\,Q) = \varnothing$ *and if $i \neq j$ then also* $\text{res-num}\,P_i \cap \text{res-num}\,P_j = \varnothing$,
8. *if P_i is labeled with* vc, *then the first block of ct_{P_i} does not occur in ct_{P_j} for $i \neq j$,*
9. *if a variable is free in a term binding in P_i, then it is either top-level quantified or free in P_i, and*
10. *ct_{P_i} and ct_{P_j} are compatible contexts.*

If a set of subgoals has a position-wellformed labeling, a rule can create fresh blocks by using numbers from the reserved numbers (condition 3). Condition 7 ensures that the new set of subgoals has also a position-wellformed labeling. Intuitively, this definition boils down to the following: a rule should always extend or keep the context of the conclusion and never modify it otherwise (condition 2). Everything new in labeling must be newly introduced by this rule. Assumptions must only be introduced if a new block is added. To avoid confusion of the user, it is also usually a good idea to use different block names in different rules.

Lemma 6.19. *Let S be a set of subgoals with a position-wellformed labeling and let R be a well-formed labeled rule. Then applying the rule R to an element of S yields again a position-wellformed labeling.*

Corollary 6.20. *Let $S = \{\text{ctxt } 0\ []\ o\ P\}$ and $\mathcal{R}$ be a set of wellformed labeled rules. Then every set S' derived from S by applications of rules from $\mathcal{R}$ has a wellformed-labeling.*

Proof. By induction, using Lemma 6.19. □

For a wellformed labeling, one can generate an appropriate cases structure. To represent the cases structure, I use ML syntax. A case is represented by the type string case,

```
datatype 'a case = Case of {
  name: 'a,
  fixes: (string * typ) list,
  asms: (string * term) list,
  binds: (string * term) list,
  cases: 'a case list
}
```

All terms in a case have the form $t = \lambda x_1 \ldots x_m.\ t'$, where the m is the number of fixed variables in this and (initially) the surrounding cases. When a case is activated, the fixed variables are applied to t for all terms t in a case and its nested cases.

Definition 6.21 (Precase). *Let $\forall x_1 \ldots x_m.\ P_1 \Longrightarrow \ldots \Longrightarrow P_m \Longrightarrow \text{vc}\ [bl_k, \ldots, bl_1]\ Q$ be a subgoal with a final wellformed labeling. Let $xs = [x_1, \ldots, x_m] \setminus \text{varss}\ [bl_k, \ldots, bl_1]$. Given a list of variables and a term t, the function* abss *defined by the following two equations abstracts t over these variables:*

$$\text{abss}[]\ t = t \qquad\qquad \text{abss}(v :: vs)\ t = \lambda v.\ \text{abss}\ vs\ t$$

Then the precase *(of type* $(\mathbb{N} \times \text{string})$ case*) of this subgoal is defined as follows:*

$$\text{precase}\ bl_i = \begin{cases} \text{C}\ \{\text{name} = \text{iden}\ bl_i, \text{fixes} = \text{vars}\ bl_i, & \\ \quad \text{asms} = asms, \text{binds} = [], \text{cases} = bl_{i+1})\} & \text{if } 1 \le i < k \\ \text{C}\ \{\text{name} = (\text{iden}\ bl_i, \text{fixes} = \text{vars}\ bl_i +\!\!+ xs, & \\ \quad \text{asms} = asms, & \text{if } i = k \\ \quad \text{binds} = (\text{«concl»}, \text{abs}\ Q) :: binds, \text{cases} = []\} & \end{cases}$$

where

$$\text{abs} = \begin{cases} \text{abss}\ (\text{vars}\ bl_1 +\!\!+ \ldots +\!\!+ \text{vars}\ bl_i) \circ \text{ut} \circ \text{uh} & \text{if } 1 \le i < k \\ \text{abss}\ (\text{vars}\ bl_1 +\!\!+ \ldots +\!\!+ \text{vars}\ bl_k +\!\!+ xs) \circ \text{ut} \circ \text{uh} & \text{if } i = k \end{cases}$$

$$\text{asm}\ P = \begin{cases} (x, \text{abs}\ P') & \text{if } \text{uh}\ P = \text{tbind}\ n\ x\ ct\ P' \\ (\text{«asm»}, \text{abs}\ P) & \text{else} \end{cases}$$

$$asms = \text{map asm}\ [P_j \mid 1 \le j \le m \wedge \text{hct}_{P_j} = [bl_i, \ldots, bl_1]]$$

$$binds = [(x, \text{abs}\ t) \mid \text{tbind}\ n\ x\ ct\ t \text{ is subterm of } P_1, \ldots, P_m, Q].$$

Moreover, ut *resp.* uh *are the functions removing* tbind *resp.* hier *labelings from the term.*

$$\begin{aligned}
&\mathsf{C}\ \{\mathsf{name} = (2, \text{«bind»}), \mathsf{fixes} = [(\text{«x»}, \mathsf{nat})], \mathsf{asms} = [], \mathsf{binds} = [],\\
&\quad \mathsf{cases} = [\,\mathsf{C}\ \{\mathsf{name} = (3, \text{«while»}), \mathsf{fixes} = [(\text{«y»}, \mathsf{nat})], \mathsf{asms} = [], \mathsf{binds} = [],\\
&\qquad \mathsf{cases} = [\,\mathsf{C}\ \{\mathsf{name} = (3, \text{«inv»}), \mathsf{fixes} = [], \mathsf{asms} = [], \mathsf{binds} = [],\\
&\qquad\quad \mathsf{cases} = [\,\mathsf{C}\ \{\mathsf{name} = (4, \text{«conseq»}), \mathsf{fixes} = [],\\
&\qquad\qquad \mathsf{asms} = A, \mathsf{binds} = B, \mathsf{cases} = []\}]\}]\}
\end{aligned}$$

where

$$\begin{aligned}
A &= [(\text{«asm»}, \lambda x\ y.\ y\ \mathsf{dvd}\ a \wedge (\forall m.\ \mathsf{odd}\ m \wedge m\ \mathsf{dvd}\ a \implies m\ \mathsf{dvd}\ y) \wedge \mathsf{even}\ y)]\\
B &= [(\text{«concl»}, \lambda x\ y.\ (y/2)\ \mathsf{dvd}\ a \wedge (\forall m.\ \mathsf{odd}\ m \wedge m\ \mathsf{dvd}\ a \implies m\ \mathsf{dvd}\ (y/2))]
\end{aligned}$$

Figure 6.4.: Precase for (6.5) from Example 6.7

A precase is the interpretation of a single labeled verification condition as a cases structure, see Figure 6.4 for an example.

The definition of a precase distinguishes between the innermost block, i.e., bl_k, and the other blocks: The precase for the innermost block contains all those assumptions and top-level quantified variables which are not explicitly labeled to belong to any of the earlier blocks.

To get from precases to a single cases structure for a well-formed goal structure, the precases must be ordered and grouped.

Definition 6.22 (Case). *Let S be a set of subgoals with a final wellformed labeling. We define an equivalence relation $\doteq$ on precases by $x \doteq y \leftrightarrow \mathsf{name}\ x = \mathsf{name}\ y$, where* name *refers to the name field of a precase. Let $\hat{S}$ be a set of representatives of this equivalence relation and $[s]_S$ be the equivalence class of some $s \in S$ w.r.t. $\doteq$ in S. Let*

$$\mathsf{sort} : (\mathsf{int} \times \mathsf{string})\ \mathsf{precase\ set} \to (\mathsf{int} \times \mathsf{string})\ \mathsf{precase\ list}$$

be the function sorting a set of precases by its name. Then the cases *of the subgoals S are* cases S*, where* cases *is defined recursively by*

$$\begin{aligned}
\mathsf{cases}\ T &= \mathsf{disambig}\ (\mathsf{map}\ (\lambda t.\ \mathsf{merge}\ t\ [t]_T)\ (\mathsf{sort}\ \hat{T}))\\
\mathsf{merge}\ t\ T &= \mathsf{C}\ \{\mathsf{name} = \mathsf{snd}\ (\mathsf{name}\ t), \mathsf{fixes} = \mathsf{fixes}\ t, \mathsf{asms} = \mathsf{asms}\ t,\\
&\qquad \mathsf{binds} = \mathsf{binds}\ t, \mathsf{cases} = \mathsf{cases}\ (\mathsf{maps\ cases}\ T)\}.
\end{aligned}$$

Here, the disambig *function takes care of the disambiguation the case names: if a list contains n cases with the same name x, they are renamed $x, x_2, \ldots, x_n$. following the order of the list.*

The merge operation replaces the fixes, assumes and binds by those of a representative and calls cases for the union of the subcases of the equivalence class of this representative. For a wellformed goal state, all cases in the same equivalence class have the same fixes (except for the names) and assumptions, so replacing the fixes is sound. Binds occur only in leaf cases

and the equivalence class of a leaf case only consists of a single case (due to condition 4 of Definition 6.16).

In Isabelle, a *proof method* is a function st → st × string case list where st is a goal state. Now we can build a proof method for $\mathcal{L}$ which supports structured proofs as described in Section 6.2.1: Use the VCG with the labeled rules to generate verification conditions with a wellformed labeling. Then, use cases to generate the cases structure and finally remove the labeling from the verification conditions by unfolding the labeling constants.

The cases are then setup so that each subgoal can be discharged by activating one of the cases, selecting the appropriate term binding and prove the resulting proof obligation.

6.4. A Labeling VCG for $\mathcal{L}$

To capture the program structure, we attach labels as defined in Section 6.3 to the verification conditions and modify the rules of the calculus to compute these labels. That way, the modifications to the VCG can be kept to a minimum: it is not necessary to add manual book-keeping. In many cases it suffices to use the modified rule set. In this section, I describe a labeled hoare calculus for the language $\mathcal{L}$.

When solving a verification condition, there are usually three things we need to consider for the proof:

1. *Where* in the program are we? (e.g., "the while loop in line 4")
2. *What* kind of obligation is this? (e.g., preservation of the loop invariant)
3. *How* did we get there? (e.g., which branches of conditional statements did we take before arriving here?)

In this section, I consider labels which describe the whole verification condition. For the VCG of $\mathcal{L}$, this corresponds to the *Where* and *What*. In Section 6.6, I will give an outlook how the *How* could also be exposed to the user.

How should we express the *Where*? My goal is that the cases in a structured proof reflect the structure of the program (see Example 6.3): when looking at the cases I want to see all statements of the program which give rise to a verification condition, in the same order and the same nesting as in the original program. $\mathcal{L}$ has a block structure: Control structures like while or cond are commands containing other commands. This structure is mapped directly to the blocks from Section 6.3.

The rule for sequential composition just passes the position after the first command to the position before the second command. The first premise introduces a new top-level quantified variable, which is recorded in a block.

$$\frac{\overbrace{\forall x.\ \{\!|R\ x|\!\}\ c_2\ x\ \{\!|Q|\!\}}^{\text{ctxt } (o'+1)\ (\langle o', \text{«bind»}, [x]\rangle_B ::ct)\ o} \qquad \overbrace{\{\!|P|\!\}\ c_1\ \{\!|R|\!\}}^{\text{ctxt } i\ ct\ o'}}{\underbrace{\{\!|P|\!\}\ \text{bind}\ c_1\ c_2\ \{\!|Q|\!\}}_{\text{ctxt } i\ ct\ o}}\ \text{Bind}_C$$

My readers may object to adding a block in Rule Bind_C: Intuitively, one does not consider sequential composition to imply any nesting. One the other hand, the lambda abstraction in bind $c_1(\lambda x.c_2\ x)$ definitely implies a nesting to the right and we can only refer to bound variable if it has a name. In Section 7.2, we will see that the blocks arising from sequential composition can be nicely integrated into an Isar proof.

Rule Return does not have any premises, so it only needs to compute the output number from the input number.

$$\frac{}{\underbrace{\{\!|P\ x|\!\}\ \mathsf{return}\ x\ \{\!|P|\!\}}_{\text{ctxt } i\ ct\ (i+1)}}\ \text{Return}_C$$

Rule Cond is more interesting: First, the output number is computed by counting the lines for the cond itself and the commands in the two branches. The two branches are nested side-by-side in cond, so both are put in separate blocks inside a context $ct' = \langle i, \text{«cond»}, []\rangle_B :: ct$. The computation of the line numbers follows the same scheme as in Rule Bind_C.

$$\frac{\text{ctxt } (i+1)\ (\langle i, \text{«then»}, []\rangle_B{::}ct')\ o' \qquad \text{ctxt } (o'+1)\ (\langle o', \text{«else»}, []\rangle_B{::}ct')\ o}{\underbrace{\{\!|\overbrace{P_1}|\!\}\ c_1\ \{\!|Q|\!\} \qquad \{\!|\overbrace{P_2}|\!\}\ c_2\ \{\!|Q|\!\}}_{\{\!|\mathsf{if}\ b\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2|\!\}\ \mathsf{cond}\ b\ c_1\ c_2\ \{\!|Q|\!\}}}_{\text{ctxt } i\ ct\ o}\ \text{Cond}_C$$

The consequence rule is not attached to a concrete command, but it is used if the (computed) precondition of the proof obligation does not match the precondition of the current command. Hence, we order the verification condition arising from the consequence rule before the current command.

$$\frac{\text{ctxt } (i+1)\ ct\ o \qquad \text{vc } (\langle i, \text{«conseq»}, []\rangle_B{::}ct)}{\underbrace{\overbrace{\{\!|P'|\!\}}\ c\ \{\!|Q|\!\} \qquad P' \implies \overbrace{P}}_{\{\!|P|\!\}\ c\ \{\!|Q|\!\}}}_{\text{ctxt } i\ ct\ o}\ \text{Conseq}_C$$

As for Rule Cond_C, the statement numbers computed by Rule Conseq_C only roughly resemble line numbers: the counter is increased to satisfy the local wellformedness conditions and preserve the ordering.

In Rule While, the labeling is used to record the loop variable, which occurs in both premises as a top-level quantified variable.

$$\frac{\text{ctxt } (i+1)\ ((i, \text{«inv»}, []){::}ct'\ x)\ o \qquad\qquad \text{vc } (\langle o, \text{«post»}, []\rangle_B{::}ct'\ x)}{\underbrace{\forall x.\ \overbrace{\{\!|I\ x \wedge b\ x|\!\}}\ c\ x\ \{\!|I|\!\} \qquad \forall x.\ I\ x \wedge \neg b\ x \implies \overbrace{P\ x}}_{\{\!|I\ x|\!\}\ \mathsf{while}\ b\ I\ c\ x\ \{\!|P|\!\}}}_{\text{ctxt } i\ ct\ (o+1)}\ \text{While}_C$$

where $ct' = \lambda x.\ \langle i, \text{«while»}, [x]\rangle_B$. Note that premises have been put into the same block «while». Recall the **case** (while n) syntax from the introductory example: this labeling allows us to treat the loop state variable consistently for both the side condition and all verification conditions arising from the Hoare triple.

For Rule WhileT, the premise

$$\forall x.\ \{\!|I\ x \wedge b\ x|\!\}\ c\ x\ \{\!|\lambda x'.\ I\ x' \wedge (x', x) \in R|\!\}$$

```
⦃True⦄
x ←WHILE (λx. snd x < a)
  INV (λx. i ≤ a ∧ fst x = snd x · b)
  BODY (λx.
    y ←WHILE (λx. snd y < b)
      INV (λx. snd x < a ∧ snd y ≤ b
        ∧ fst y = snd x · a + j)
      BODY (λx. return (fst y + 1, snd y + 1))
      START (fst x, 0);
    return (fst y, snd x + 1)
  START (0,0);
return (fst x)
⦃λr. r = a · b⦄
```

(a) Selectors

```
⦃True⦄
(r, _) ←WHILE (λ(r, i). i < a)
  INV (λ(r, i). i ≤ a ∧ r = i · b)
  BODY (λ(r, i).
    (r, _) ←WHILE (λ(r, j). j < b)
      INV (λ(r, j). i < a ∧ j ≤ b
        ∧ r = i · a + j)
      BODY (λ(r, j). return (r + 1, j + 1))
      START (r, 0);
    return (r, i + 1)
  START (0,0);
return r
⦃λr. r = a · b⦄
```

(b) Pattern Matching

Figure 6.5.: $\mathcal{L}$ program for multiplication by repeated increment

basically contains two proof obligations: invariant preservation and termination. As these will finally end up in a verification condition, it is useful to present them to the user separately. Labeling this premise as

$$\forall x.\ \overbrace{\{\!|I\ x \wedge b\ x|\!\}\ c\ x\ \{\!|\lambda x'.\ \underbrace{x'}_{\text{tbind «inv» } i} \wedge \underbrace{(x', x) \in R}_{\text{tbind «var» } i}|\!\}}^{\text{ctxt } i+1\ (i, \text{«inv»}, [])::ct'\ x\ o}$$

and the rest as in Rule While$_C$, the resulting VC will not only contain the default term binding for the conclusions, but also term bindings called "inv" and "var" corresponding to $I\ x'$ and $(x', x) \in R$ (which might have been transformed by the VCG).

It is easy to see that this set of labeled rules conforms to the local wellformedness conditions. If we change the VCG for $\mathcal{L}$ to use the labeled rules, it produces the labeled verification conditions from Example 6.7 for the program from Example 6.2.

6.5. Splitting Tuples

Consider the while combinator for the language $\mathcal{L}$: this combinator allows only for one variable as loop state. Similarly, the branches of the cond combinator can only "modify" one variable. Multiple variables can be simulated by using tuples. To allow a more readable presentation, programs written in one of AutoCorres's languages (which are the basis for $\mathcal{L}$) make heavy use of the pattern-matching syntax $\lambda(x_1, \ldots, x_n).\ t\ x_1\ \ldots\ x_n$, cf. Figure 6.5. Here $(x_1, \ldots, x_n)$ is a shortcut for $(x_1, (x_2, \ldots, x_n))$ and $\lambda(x_1, x_2).\ t\ x_1\ x_2$ abbreviates $\lambda x.$ case-prod $(\lambda x_1\ x_2.\ t\ x_1\ x_2)\ x$. where x does not occur in t. The function case-prod is defined by case-prod $f\ (x, y) = f\ x\ y$.

To handle this syntax, the inference system for $\mathcal{L}$ is extended with Rule Product, see Figure 6.6. This rule moves case-prod out of the way – otherwise, no rule of our calculus would apply to the body of one of the while loops in Figure 6.5b. The rule is demonstrated by the partial proof

$$\frac{\forall y, z.\ x = (y, z) \implies \{\!|P\ y\ z|\!\}\ c\ y\ z\ \{\!|Q|\!\}}{\{\!|\text{case-prod}\ P\ x|\!\}\ \text{case-prod}\ c\ x\ \{\!|Q|\!\}}\ \text{Product}$$

Figure 6.6.: Inference Rule for Pattern Matching

$$\begin{aligned}
&\forall x\ r\ i\ x'.\ x = (r, i)\\
&\quad\implies \text{case-prod}\ (\lambda r\ j.\ i < a \wedge j \le b \wedge r = i \cdot b + j)\ x' \wedge \neg\text{case-prod}\ (\lambda r\ j.\ j < b)\ x'\\
&\quad\implies \text{vc}\ \langle 6, \text{«post»}, []\rangle_B\ [\langle 3, \text{«while»}, [x']\rangle_B, \langle 1, \text{«invariant»}, []\rangle_B, \langle 1, \text{«while»}, [x]\rangle_B]\\
&\qquad\text{case-prod}\ (\lambda x\ y.\ \text{case-prod}\ (\lambda r\ i.\ i \le a \wedge r = i \cdot b)\ (r, i + 1))\ x'
\end{aligned}$$

Figure 6.7.: A VC of Figure 6.5b. The premise $x = (r, i)$ arose from Rule Product (lifted to ignore the labeling). Note that x is only used in the label; the conclusion uses i instead. The outer case-prod arose from the postcondition of the inner while loop.

below (the irrelevant branches are omitted for space reasons):

$$\cfrac{\cfrac{\cfrac{\forall x, y, z.\ x = (y, z) \implies \{\!|I\ (y, z) \wedge b\ (y, z)|\!\}\ c\ y\ z\ \{\!|I|\!\}}{\forall x.\ \{\!|\text{case-prod}\ (\lambda y\ z.\ I\ (y, z) \wedge b\ (y, z))\ x|\!\}\ \text{case-prod}\ (\lambda y\ z.\ c\ y\ z)\ x\ \{\!|I|\!\}}\ \begin{matrix}\text{Product}\\ \text{Conseq}\end{matrix}}{\forall x.\ \{\!|I\ x \wedge b\ x|\!\}\ \text{case-prod}\ (\lambda y\ z.\ c\ y\ z)\ x\ \{\!|I|\!\}}\ \text{While}}{\{\!|I\ r|\!\}\ \text{while}\ b\ I\ (\lambda(y, z).\ c\ y\ z)\ r\ \{\!|Q|\!\}} \tag{6.15}$$

A mechanism for structured proofs should support this style of simulating multiple variables. In particular, this means the ability to write **case** (while r i) for a while loop using a tuple as loop state, as in Figure 6.5b, and have r and i correspond to the expected values both in the VC belonging to the invariant and the VC belonging to the postcondition. There are two separate issues:

- There is a term case-prod $t\ x$ in the premises or conclusion of a verification condition, where x is a top-level quantified variable. This x needs to be replaced with fresh bound variables y, z and occurrences of case-prod $t\ x$ need to be replaced by $t\ y\ z$.
- The command of a Hoare triple has the form case-prod $t\ x$. Then, Rule Product removes the pattern matching and introduces additional bound variables y and z, without removing x. The labeling needs to record that x was replaced by (y, z).

The first case occurs e.g. for the postcondition of Rule While, the second for the invariant in the same rule (see Figure 6.7). Note that both cases need to be handled consistently and simultaneously for all verification conditions to preserve the wellformedness of the labeling.

Only the second issue needs a change to the labeling. We do not yet have a labeling to express that the block which recorded the variable x should now record y, z instead. One could rewrite the label with the premise $x = (y, z)$, but for the sake of robustness it is preferable not to rewrite with arbitrary equations. Hence, we introduce a dedicated labeling constant:

Definition 6.23 (Tuple Label). *The labeling constant* split *takes two terms of the same type and is defined as* split $a\ b \coloneqq (a = b)$.

This label differs from the ones in Section 6.4 in that it is not purely syntactic and is used as an additional assumption, not attached to the conclusion. Similar to ctxt, split only occurs in intermediate steps and is not expected to end up in the final verification conditions.

Note that Rule Product introduces two variables. These will not be recorded in a block, as they only exist temporarily, until the split is resolved. Instead, they are marked with a tuple label.

Definition 6.24 (Wellformed Split-Labeling). *Let S be a set of subgoals having a position-wellformed labeling such that for each $\varphi \in S$ holds: for any assumption* split $x\ y$ *in φ, y is a (tuple of) top-level quantified variables and the variables in y occur neither on the right hand side of another* split *nor in the blocks of the labeling of this subgoal. Then S is a* position-wellformed split-labeling.

A wellformed labeled rule is a wellformed split-labeled rule*, if for each assumption* split $x\ y$ *in a premise P holds: y is a (tuple of) top-level quantified variables in P and the variables in y occur neither on the right hand side of another* split *nor in* blocks $P \setminus$ blocks Q*, where Q is the conclusion of the rule.*

The labeled version of Rule Product then is defined as follows:

$$\frac{\forall y, z.\ \text{split}\ x\ (y, z) \implies \overbrace{\{\!|P\ y\ z|\!\}\ c\ y\ z\ \{\!|Q|\!\}}^{\text{ctxt}\ i\ ct\ o}}{\underbrace{\{\!|\text{case-prod}\ (\lambda y\ z.\ P\ y\ z)\ x|\!\}\ \text{case-prod}\ (\lambda y\ z.\ c\ y\ z)\ x\ \{\!|Q|\!\}}_{\text{ctxt}\ i\ ct\ o}}\ \text{Product}_C$$

Obviously, this rule is a wellformed split-labeled rule. I now describe a transformation from a position-wellformed split-labeling into a position-wellformed labeling which eliminates the tuples recorded in the labeling:

Definition 6.25 (Tuple Postprocessing). *For all labeled verification conditions, perform the following steps:*

1. *Split all variables of tuple-type which are recorded in labeling. This is easily achieved by controlled rewriting with the theorem*

   $$\forall x.\ P\ x \equiv \forall a\ b.\ P\ (a, b).$$

   Update the labeling by flattening the tuples.[2]

2. *Rewrite all occurrences of* case-prod $(\lambda z\ w.\ t\ z\ w)\ (x, y)$ *to $t\ x\ y$.*

3. *Simplify* split *annotations by replacing each premise of the form* split $(x, y)\ (z, w)$ *by two premises* split $x\ z$ *and* split $y\ w$.

[2] If the program uses tuples not only for simulating multiple variables, this step might split more then intended. An easy solution is the introduction of special tuple type.

4. *For each annotation* split $x\ y$ *where* x *is top-level quantified, rewrite* x *to* y *and remove the annotation.*[3]

Lemma 6.26 (Tuple Postprocessing Preserves Wellformedness). *If S is a set of subgoals with a position-wellformed split-labeling, then the result of the tuple postprocessing of S also has a position-wellformed labeling.*

Proof. Steps 1 to 3 obviously preserve the wellformedness. After step 3, in each annotation split $x\ y$ either x is not a top-level quantified variable or x is a top-level quantified variable and not of tuple type. In the former case, the unfolding of split does not affect the labeling (and hence the wellformedness). In the latter case, both x and y are top-level quantified variables of the same type, where y is not bound in the blocks of the labeling; so the rewriting preserves the wellformedness. □

After Tuple Postprocessing, the verification conditions of Figure 6.5b faithfully present the "multiple variables" view.

Example 6.27. *Recall the verification condition from Figure 6.7. By Tuple Postprocessing this is transformed to:*

$$\begin{aligned}&\forall r\ i\ y\ y'.\ (i < a \wedge y' < b \wedge y = i \cdot b + y') \wedge \neg y' < b\\ &\quad \Longrightarrow \mathrm{vc}\ \langle 6, \text{«post»}, []\rangle_B\ [\langle 3, \text{«while»}, [y, y']\rangle_B, \langle 1, \text{«invariant»}, []\rangle_B, \langle 1, \text{«while»}, [r, i]\rangle_B]\\ &\qquad (i + 1 \leq a \wedge y = (i + 1) \cdot b)\end{aligned}$$

After generating the cases, if the user selects the nested cases

case (while _ i) { **case** invariant { **case** (while r j) { **case** post ... }}}

then assumption and conclusion are presented as

$$(i \leq a \wedge j \leq b \wedge r = i \cdot b + j) \wedge \neg j < b \qquad \text{and} \qquad i + 1 \leq a \wedge r = (i + 1) \cdot b,$$

respectively, which is exactly what the notation in Figure 6.5b suggests.

6.6. Other Applications

So far, we have considered labeling the verification conditions generated from a program. In this section, I give two further examples for the labeling: a small verification condition generator for logical formulas built from if-then-elses and atoms and a replacement for Isabelle's unfold_locales proof method.

Example 6.28 (Decomposing Conditionals). *A labeling VCG for logical formulas made from if-then-else and atoms can be constructed from the following rules.*

[3] This step is easily implemented by instantiating Isabelle's hypsubst tactic for the split constant.

As a shortcut we write $ct_t = \langle i, \text{«then»}, [] \rangle_B :: ct$ *and* $ct_e = \langle i, \text{«else»}, [] \rangle_B :: ct$.

$$\frac{\overbrace{\overbrace{t}^{\text{vc } ct}}^{\text{ctxt } i \text{ } ct \text{ } (i+1)}}{\text{Final}_C} \qquad \frac{\text{hier } ct_t\ a \implies \overbrace{b}^{\text{ctxt } (i+1) \text{ } ct_t \text{ } o'} \quad \text{hier } ct_e\ (\neg a) \implies \overbrace{c}^{\text{ctxt } (o'+1) \text{ } ct_e \text{ } o'}}{\underbrace{(\text{if } a \text{ then } b \text{ else } c)}_{\text{ctxt } i \text{ } ct \text{ } o}} \text{If}_C$$

The VCG decomposes the goal with Rule If_C *and finishes the computation of the statement numbers by applying Rule* Final_C *to each subgoal. The* hier *label records where the assumption was introduced, so the assumption ends up in the case described by* ct_t *resp.* ct_e.

Applying this VCG to the expression

$$\text{if } a \text{ then } (\text{if } b \text{ then } c \text{ else } d) \text{ else } e$$

yields the following list of cases:

```
C {name = «then», fixes = [], asms = [(«asm», a)], binds = [],
   cases = [ C {name = «then», fixes = [], asms = [(«asm», b)], binds = [(«goal», c]},
             C {name = «else», fixes = [], asms = [(«asm», ¬b)], binds = [(«goal», d)]}]},
C {name = «else», fixes = [], asms = [(«asm», ¬a)], binds = [(«goal», e)]}
```

This VCG can serve as a basis to expose the various execution paths of a program, i.e., the *How* of Section 6.4: the VCG for $\mathcal{L}$ described in Section 6.4 does not generate separate verification conditions for the two branches of a conditional command. Instead, the weakest precondition of the conditional command contains a conditional expression, which will usually end up as the conclusion for a verification condition generated by Rule Conseq.

The use of annotated terms to generate a cases structure is not only useful for program verification. Another candidate is Isabelle's unfold_locales proof method (and its cousins intro_locales and intro_classes). This method is used to prove that some set of parameters constitute an instance of a locale. The result of this proof method are subgoals for each of the assumptions of the locale and the assumptions of the ancestors of the locale. As all these have names, it would be useful to have these names reflected in a cases structure. However, this is deemed too complicated to be worth the effort by the Isabelle developers:

> *There are presently two main applications where goal cases are really needed in regular Isar proofs: intro_classes and intro_locales. Since these proof methods are a bit too complex to provide properly named cases (although it is possible in theory), post-hoc addition of "goals" is the way to do it.*
>
> — Makarius Wenzel, 2015-07-02 on the Isabelle development list

In this quote, "goal cases" refers to the technique of generating cases directly from (unlabeled) subgoals by naming them $\text{«goal}_1\text{»}$, $\text{«goal}_2\text{»}$ and so on.

Conceptually, unfold_locales is very simple: It applies a set of introduction rules repeatedly, which can be written as intro *rules* in Isabelle. With the existing infrastructure, this repeated

application suffices to make building a cases structure to complex: instead of using a basic combinator like intro, one would need to apply the rules step-by-step, building the cases structure simultaneously. Using the labeling presented in this chapter makes this complexity go away.

Example 6.29 (Labeled unfold_locales). *Let $\mathcal{R}$ be the set of theorems used by unfold_locales and assume that for each rule $R \in \mathcal{R}$ there is a naming function f_R such that $f_R\ i$ is the name of the i-th premise of R. We write $ct_i = \langle o_{i-1}, f_R\ i, [] \rangle_B :: ct$. Then the labeling of such a rule is defined as*

$$\mathsf{lab}\, R = \frac{\overbrace{P_1}^{\text{ctxt } o_0\ ct_1\ o_1} \quad \ldots \quad \overbrace{P_m}^{\text{ctxt } o_{m-1}\ ct_m\ o_m}}{\underbrace{Q}_{\text{ctxt } o_0\ ct\ o_m}} \quad \textit{where} \quad R = \frac{P_1 \quad \ldots \quad P_m}{Q}$$

Then, a version of unfold_locales which produces proper case names can be defined by applying the rules lab $\mathcal{R}$ *repeatedly (with* intro*, as usual), finishing the labeling by applying Rule* Final_C *to all resulting subgoals, and then applying the procedure from Definition 6.22.*

6.7. Discussion

In this chapter, I described a method to expose the implicit structure of verification conditions to the user in the Isabelle/HOL theorem prover. This method consists of two parts: First, by labeling the rules the VCG uses, the structure is explicitly recorded in the generated verification conditions. Second, these labels are turned into a cases structure which is presented to the user. The first part requires only minor changes to the VCG, while the second part is completely independent. For the user, the advantages are twofold: The user can directly apply her intuition to a verification condition, without needing to interpret it first, and the full power of the Isar proof language is available. Equally relevant, the proofs become easier to read and hence to maintain. A case study using the case labeling to verify a non-trivial program is given in Chapter 7.

This method of labeling is not restricted to program verification as demonstrated in Section 6.6. Currently, the cases mechanism in Isabelle is only used by a few, specialized methods as creating meaningful cases with the primitives provided by Isabelle is tedious. My labeling makes it feasible to use cases more widely.

I have implemented the second part, i.e., the procedures described in Definition 6.22 and Definition 6.25, in Isabelle/HOL. For the "Hoare" theory in the Isabelle distribution and the languages used by AutoCorres, I provided labeled VCGs (i.e., an implementation of the first part). The latter has been used to verify the Planarity Checker in Chapter 7 and proved to be very useful there. The implementation, together with labeled VCGs for "Hoare" and a simplified version of the AutoCorres languages can be found in the Archive of Formal Proofs [62] (the full version depends on AutoCorres, which is not yet in the AFP).

The act of labeling the rules of the VCG is straightforward. A reason for that is that each rule can be labeled in isolation, obeying the wellformedness conditions of Definition 6.18 and Definition 6.24. In particular, the labeling of the rules does not need to be reconsidered if additional rules are added to the calculus.

In this chapter, I assumed that the VCG only applies a limited set of rules to decompose a Hoare triple into its verification conditions. However, often VCGs perform additional steps, for example by automatically eliminating technical side conditions or simplifying certain patterns. Technically, any proof step preserving the wellformedness of the labeling is allowed. Additional care is necessary if subgoals are duplicated or top-level quantified variables are replaced (although I did not encounter this issue with the VCGs I adapted to use case labeling). The bigger issues lie on the user interface side: The form of the verification conditions should be predictable. This means that systematic simplifications (like the removal of case-prod or the removal of trivial verification conditions) are usually fine, but more heuristic steps (like running the standard rewriting tactic on each goal) are better avoided, as they make the proofs more brittle and confusing.

Proof tools used by a VCG often expect the subgoals to be in a certain shape, which is hidden by the labeling. Provided such a tool does not create additional subgoals, this can be usually solved by moving the label temporarily to the assumptions of the subgoal.

The labeling stores the structure in the verification conditions itself. As a result of this, the verification conditions do not need to be produced by a single monolithic proof method. Instead, arbitrary proof methods can be combined, provided they preserve the wellformedness of the labeling. This is the case for many of the common proof methods in Isabelle. This makes it easy to add extensions like the Tuple Postprocessing.

One of the guiding principles of Isar is to reduce implicit state in proofs. In particular, assumptions and subgoals should typically be explicitly spelled out in a structured proof. The cases mechanism trades explicitness for more formal structure, on the premise that it should only be used for proof methods with a predictable result.

One can claim with some merit that a VCG stretches the idea of predictable results. If one is of that opinion, it is still possible to avoid the term bindings and state the assumptions explicitly, e.g., by using literal fact propositions [78, §3.3.7] Otherwise, named assumptions and term bindings spare the author from copy-and-pasting long terms, while providing at least a rough idea of what they are referring to. In any case, case labeling creates an intelligible proof structure, which allows for a more convenient development of proofs and increases maintainability.

We conclude this chapter by reviewing some related work. As mentioned in the introduction, specialized program verification tools use labels to relate verification conditions and source positions [8, 46]. The VCG in ESC/Java [46] computes where to insert the labels, but the position defined by the label is directly taken from source program.

In an unpublished article, Gast [29] uses a similar term labeling for the verification of C-like programs in Isabelle/HOL. He provides a special command which takes a C-like program, embeds it into the logic and starts the verification process. The labels refer to positions in the AST of the program and an additional user interface shows the source code, highlighting the positions in the label. This approach relies on a tight integration between the parser, the proof rules and the user interface. In contrast, my approach does not need additional infrastructure and can be integrated with any language embedded in Isabelle with little effort.

Recently, Daniel Matichuk et al. have been working on Eisbach, an extension to the Isar proof language [49]. In particular, this language allows a direct access to the components of a subgoal, without using a classical structured proof. This allows to structure the proof of each verification condition, but does not help in exposing the program structure. In a sense, Eisbach

covers some middle-ground between the two extremes of unstructured and structured proofs I described in Section 6.2.

Moreover, a VCG often generates large subterms, for example for modified program state. In many proofs, the user will need to refer to these terms, so these should be made available to the user as term bindings. In his work on implementing separation logic on top of Simpl [71], Gast [28] extracts such terms. It remains to be seen whether this can be done in a way which keeps the generation of nested cases agnostic of the VCG.

For another approach of structuring program verification proofs, albeit with a different focus, see the work on Ribbon Proofs [80].

7. A Checker for Planarity

In Chapter 5, I discussed a non-planarity certificate and presented a verified implementation of the predicate $\mathcal{W}_{\mathsf{False}}$ together with a proof of the witness property. In this chapter, I complete the checker for the planarity test by verifying an implementation of $\mathcal{W}_{\mathsf{True}}$. However, the main focus is on the tool used for the verification, the case labeling from Chapter 6.

After sketching the implementation and the correctness arguments, I look at the verification process and the resulting proof text and discuss the advantages and short-comings of the case labeling, compared to a VCG without support for structured proofs.

7.1. Implementation

As mentioned, the main focus of this chapter is on the evaluation of the case labeling. For this, it is not necessary to start from a C implementation of the checker. Most technicalities stemming from C can already be dealt with in an abstraction step as described in Section 5.4.1. Hence, the checker is directly implemented in one of the target languages of AutoCorres, using Isabelle datatypes like natural numbers, sets, and pairs. The level of abstraction is similar to the one in the Simpl implementation of the non-planarity checker.

The used language is very similar to the language $\mathcal{L}$ described in Section 6.1, but supports a program heap (which is not actually used here). The set of labeled rules for the VCG is described in Appendix A.

Recall the definition of the witness predicate for planarity:

$$\mathcal{W}_{\mathsf{True}} := \{(x, y, w) \mid y = \mathsf{True} \wedge w \text{ is a planar map for} x\}$$

If $\mathcal{W}_{\mathsf{True}}\,(x, y, w)$ holds, then x is combinatorially planar (this can be easily seen by unfolding the definition of combinatorial planarity). For the implementation, we use the same representation of graphs as in Chapter 5, see Figure 5.4. A map is represented by three functions, corresponding to π, ρ and ρ^{-1}. See Figure 7.1 for details.

We assume that there is already a trustworthy program computing the number of components. Computing the components of a graph is a separate problem, for which again exist certifying algorithms [52, §7.4]. Alkassar et al. [2] formalize a partial checker for this problem.

The implementation consists of a procedure is-planar which can be divided into four parts:

1. test whether M is a map for G,
2. compute the number of isolated nodes,
3. compute the number of face cycles, and
4. check whether the genus is 0.

record IMap = im-rev : $\mathbb{N} \to \mathbb{N}$, im-succ : $\mathbb{N} \to \mathbb{N}$, im-pred : $\mathbb{N} \to \mathbb{N}$

definition
mk-map G M := (G, perm-restrict (im-rev M) A_G, perm-restrict (im-succ M) A_G)
definition as-bidir (G, π, ρ) := (G, π)

Figure 7.1.: Representation of maps in the implementation. The function mk-map converts an IMap into a map and as-bidir treats a map as bidirected digraph.

For this procedure, I prove the following correctness result:

Theorem 7.1 (is-planar implements $\mathcal{W}_{\mathsf{True}}$). *Let iG be a representation of a finite digraph G with a distinct list of vertices and let c be the number of strongly connected components of G. Then* is-planar *iG iM c terminates. It returns* True *if iM is a representation of a planar map for G and* im-pred *iM is the inverse of* im-succ *iM on the arcs of iG. Otherwise, it returns* False.
In particular, is-planar *returns* True *only if G is a planar.*

Of the steps above, step 1 is the most involved part of the proof. As I can demonstrate all my points about case labeling on this single step, I will not go into detail on the remaining parts. Below, I shortly discuss the implementation and verification of the procedure is-map, implementing Item 1. The full implementation and proof is available in the Archive of Formal Proofs [64].

Map Property Recall that (G, π, ρ) is a map if π reverses the arcs of G and ρ permutes the outgoing arcs of each vertex cyclically. Figure 7.2a shows the implementation of the is-map procedure checking whether (G, π, ρ) = mk-map G M for some IMap M is a map.

The first loop tests whether π reverses the arcs of G, i.e., whether (G, π) is a bidirected digraph. The remainder of the is-map procedure checks whether ρ has the required properties. The second loop tests whether ρ is a permutation of A_G, mapping arcs to other arcs with the same tail. Here, im-pred iM is a witness inside a witness: It allows to certify easily that ρ is injective. As $\rho : \mathsf{A}_G \to \mathsf{A}_G$ and A_G is finite, bijectivity follows.[1]

If the check implemented by the second loop is successful, we already know that orbit $\rho\ a \subseteq$ out-arcs $G\ a$ for all $a \in \mathsf{A}_G$. The last loop then checks whether $\subseteq$ can be replaced by equality. The idea is to iterate over the arcs and mark both its tail and all arcs in its orbit. If an unmarked arc is encountered whose tail is already marked, the orbit is a proper subset.

The invariants for the first two loops are straightforward, see Figure 7.3 for the invariants of the last loop. It is then straightforward to verify that is-map returns True if and only if (G, π, ρ) is a map and ρ is the inverse of ρ^{-1}. Note that the termination of the inner loop of the third loop depends on ρ being a function $\mathsf{A}_G \to \mathsf{A}_G$, which is why $succ_1$ is part of the loop condition.

[1] A certifying algorithm producing the certificate will usually be able to produce the inverse of ρ. For example, LEDA stores the order of the arcs around a vertex as a cyclic list. From this representation, both im-pred iM and im-succ iM are easily computed.

```
definition is-map iG iM ≔
 DO (_, rev) ← while
   (λ(i, ok) s. i < |ig-es iG| ∧ ok)
   (λ(i, ok). DO
     j ← return (im-rev iM i);
     in ← return (j < |ig-es iG|);
     neq ← return (j ≠ i);
     swp ← return (ig-es iG ! j = swap (ig-es iG ! i));
     invol ← return (im-rev iM j = i);
     return (i + 1, in ∧ neq ∧ swp ∧ invol)
    OD)
   (0, True);
  (_, succ1) ← while
   (λ(i, ok) s. i < |ig-es iG| ∧ ok)
   (λ(i, ok). DO
     j ← return (im-succ iM i);
     in ← return (j < |ig-es iG|);
     ends ← return (ig-tail iG i = ig-tail iG j);
     perm ← return (im-pred iM j = i);
     return (i + 1, in ∧ ends ∧ perm)
    OD)
   (0, True);
  (_, succ2, _, _) ← while
   (λ(i, ok, V, A) s. i < |ig-es iG| ∧ succ1 ∧ ok)
   (λ(i, ok, V, A). DO
     (x, V, A) ← condition (λ_. ig-tail iG i ∈ V)
      (return (i ∈ A, V, A))
      (DO
       (A', j) ← while
        (λ(A', j) s. j ∉ A')
        (λ(A', j). DO
          A' ← return (insert j A');
          j ← return (im-succ iM j);
          return (A', j)
         OD)
        (∅, i);
       V ← return (insert (ig-tail iG j) V);
       return (True, V, A ∪ A')
      OD);
     return (i + 1, x, V, A)
    OD)
   (0, True, ∅, ∅);
  return (rev ∧ succ1 ∧ succ2)
 OD
```

(a) Implementation

```
{ case weaken then show ?case
   by (auto simp: rev_inv_def) }
{ case (while i ok)
 { case invariant
  { case weaken then show ?case
     by (auto simp: rev_inv_def elim: less_SucE) }
 }
 { case wf show ?case by auto }
 { case postcondition ... }
}
case (bind _ rev)
{ case (while i ok)
 { case invariant
  { case weaken ... }
 }
 { case wf show ?case by auto }
 { case postcondition then show ?case
    by (auto simp: final_def succ1_inv_def) }
}
case (bind i_succ1 succ1)
{ case (while i ok V A)
 { case invariant
  { case weaken
    interpret postcond0 iG iM rev i_succ1 succ1
     using weaken by unfold_locales auto
    have i < |ig-es iG| and succ1 ok
     using loop_cond by auto
    ...
    show ?case
    proof branch_casify
     case then ...
    next
     case else ...
    qed
  }
  { case if case else case (while A' i')
   { case invariant
    { case weaken ... }
   }
   { case wf show ?case by simp }
   { case postcondition ... }
  }
 }
 { case wf show ?case by auto }
 { case postcondition ... }
```

(b) Partial proof

Figure 7.2.: The is-map procedure.

definition succ$_2$-inv iG iM rev i_{succ_1} $succ_1$ i ok V A ≔
A = (⋃k < (if ok then i else $i - 1$). orbit (im-succ iM) k)
∧ V = {ig-tail iG k | k. k < (if ok then i else $i - 1$)}
∧ ok = (∀k < i. ∀j < i. ig-tail iG k = ig-tail iG j ⟶ j ∈ orbit (im-succ iM) k)
∧ rev = bidirected-digraph (as-bidir (mk-map (mk-graph iG) iM))
∧ succ$_1$-inv iG iM i_{succ_1} $succ_1$ ∧ final iG i_{succ_1} $succ_1$
∧ i ≤ |ig-es iG| ∧ wf-digraph (mk-graph iG))

definition succ$_2$-inner-inv iG iM i_{succ_1} $succ_1$ i ok V A i' A' ≔
A' = (if $i = i' \wedge i \notin A'$ then ∅ else {i} ∪ segment (im-succ iM) i i')
∧ $i' \in$ orbit (im-succ iM) i
∧ ig-tail iG $i \notin V$ ∧ $succ_1$ ∧ ok ∧ succ$_2$-inv iG iM rev i_{succ_1} $succ_1$ i ok V A ≔
∧ i < |ig-es iG|

Figure 7.3.: Invariants of the final loop of is-map and its inner loop. The variable names correspond to those used in Figure 7.2b. The if-then-else in the first invariant covers the early exit of the loop (A and V are not updated in this case). The if-then-else in the second invariant is necessary as $i = i'$ holds at the beginning (where $A' = \varnothing$) and the end of the loop (where $A' =$ orbit (im-succ iM) i)).

7.2. Evaluation of the Case Labeling

Proof Layout Figure 7.2b shows an excerpt of the correctness proof for the is-map procedure. Before applying the VCG, the program has been annotated with the invariants (Figure 7.3). For this, the rewrite method described by the author and one of its students [74] has been used. I want to draw my reader's attention to a few fine points regarding the layout of the proof.

As discussed in Section 6.4, the rule for bind opens a new block. While this is necessary to capture the variables introduced by bind, it results in deeply nested cases, instead of a flat list of cases. Luckily, these blocks do not need to be reflected explicitly in the Isar proof text: The **case** command activates the components of a case for the rest of the proof block, without the need to open a new proof block. The layout in Figure 7.2b takes advantage of this to present the linear structure of the program: all cases for bind are activated in the outermost proof block (see lines 11 and 20).

This layout works nicely together with the folds the prover IDE Isabelle/jEdit automatically creates for the proof text: in the editor, each block wrapped in curly braces can be collapsed to a single line, showing only the contexts of the first line. Hence, if the innermost proof blocks (containing the actual proofs for the verification conditions) are collapsed, the proof structure is clearly visible in the editor.

This allows it to prove even complicated verification conditions directly in this structured proof without compromising legibility. In contrast, for unstructured proofs it is paramount to separate the proof of complicated verification conditions into dedicated lemmas. In the verification of is-map I used a split approach: Partial results which would be needed in more then one case where moved into separate lemmas or a locale.

In Isar, it is customary to separate cases with the **next** command, which closes the current

block and opens a new proof block. For the reasons detailed above, it is preferable to use explicit curly braces instead.

The implementation does not bind the first result of the second loop (see line 13 in Figure 7.2a). Nevertheless, the user can still bind this result in the proof (compare i_{succ_1} in line 20 in Figure 7.2b). For example, the variable i_{succ_1} is used the invariant.

To prove that the final loop preserves its invariant, one has to consider two code paths, depending on which branch of the conditional statement has been taken taken. Hence, the conclusion of the associated case weaken in line 23 is a conditional expression. Here, I used the labeling for conditionals from Section 6.6 to prove both branches separately (lines 29-34).

Leaving aside this sub-proof, only one branch of the conditional statement is visible at all in the proof structure of the program (line 36). The reason is that only one branch contains a command (namely the inner while loop) giving rise to a verification condition.

Proof Strategy Experience shows that verification of programs is not a linear process. Instead, it is an explorative process where loop invariants, the program and sometimes even the specification are adapted as the ongoing proof uncovers problems.

Case Labeling supports this process: as selecting the desired verification condition can be done with almost no effort, it encourages a non-linear proof style. For example, in the verification of is-map, the first step was devising the invariants of the three outer loops.[2] Knowing the VCG, I could then identify trivial verification conditions by their name and discharge these, often without looking at the formulas (e.g., lines 1, 5, and 17).

After that, I proved the postcondition of is-map, before even thinking about a formal invariant of the inner loops. This style of proving allowed me to keep my mind on one aspect of the program at a time, without getting lost in details.

Changing the program or the invariants often breaks already finished parts of the proof. With Case Labeling, it becomes immediately visible which parts of the proof are affected. In the verification of is-map, this helped the author a few times to detect trivial mistakes, before spending time on a wrong invariant in other parts of the proof.

Program Variables Apart from naming the verification conditions, case labeling provides the following features to facilitate writing intelligible structured proofs: Predictable naming of variables, splitting variables of product type, named assumptions, and term bindings.

When a case is activated, the bound variables can be named in a controlled manner. Having predictable names for the bound variables greatly reduces the mental effort necessary to parse the verification conditions. The user is free to use meaningful names, the same names as in the program, or to mark them as irrelevant by not giving them any name. In any case, relating variables in proof and program only needs a look at the proof resp. program structure. The labeled rules given in Appendix A bind variables in the outermost case possible. Using the proof layout recommended in this section then guarantees a consistent naming of variables across verification conditions.

[2] Actually, the very first step is usually writing down a skeleton of the structured proof. For this, the command print_nested_cases supplied by the implementation of case labeling is useful. It shows the full case structure generating by the case labeling (Isabelle's standard print_cases command only shows the outermost cases).

As a result, in the verification of is-map, it its obvious for both the author and any later reader of the proof that assuming $succ_1$ means that the second loop delivered a positive result. This information is obtained by just looking at the single **case** command introducing this variable and its position in the proof structure (see line 20). To understand that $succ_1$ refers to the second loop variable of the second loop, one does not even need to know the program!

Splitting product variables in separate variables is a prerequisite for taking full advantage of predictable names. It relieves the user of manually splitting the products and prevents the automatic splitting by various proof tools, which would lose the predictability of names. By simplifying case-prod, it makes assumptions and goals shorter and therefore more intelligible.

Example 7.2. *Consider the verification condition for the postcondition of the last while loop (i.e., the case in line 45). With case labeling, the user gets the theorems*

succ-orbits-inv *iG iM i ok V A*
rev = bidirected-digraph (as-bidir (mk-map (mk-graph *iG*) *iM*))
succ-perm-inv *iG iM* i_{succ_1} $succ_1$
final *iG* i_{succ_1} $succ_1$
$i \leq$ |ig-es *iG*|
wf-digraph (mk-graph *iG*)
$\neg$ ($i <$ |ig-es *iG*| $\wedge$ $succ_1 \wedge ok$)

to prove the following goal:

($rev \wedge succ_1 \wedge ok$) = (digraph-map (mk-map (mk-graph *iG*) *iM*)
$\wedge$ ($\forall i <$ |ig-es *iG*|. im-pred *iM* (im-succ *iM i*) = *i*))

Here, variable names correspond to the names fixed by the user. Without case labeling, the following goal state is shown to the user:

$\forall r\ i\ rev\ ra\ ia\ succ_1\ rb\ s.$
$r = (i, rev) \Longrightarrow$
$ra = (ia, succ_1) \Longrightarrow$
(case *rb* of $(i, a) \Rightarrow$ case *a* of $(ok, a) \Rightarrow$ case *a* of $(V, A) \Rightarrow$
λs. succ-orbits-inv *iG iM i ok V A* $\wedge$
rev = bidirected-digraph (as-bidir (mk-map (mk-graph *iG*) *iM*)) $\wedge$
succ-perm-inv *iG iM ia* $succ_1$ $\wedge$
final *iG ia* $succ_1$ $\wedge$
$i \leq$ |ig-es *iG*| $\wedge$ wf-digraph (mk-graph *iG*))
s $\Longrightarrow$
$\neg$ (case *rb* of $(i, a) \Rightarrow$ case *a* of $(ok, a) \Rightarrow$ case *a* of $(V, A) \Rightarrow$
$\lambda s.\ i <$ |ig-es *iG*| $\wedge$ $succ_1 \wedge ok$)
s $\Longrightarrow$
(case *rb* of $(x, xb, xa, y) \Rightarrow$
$\lambda a.$ ($rev \wedge succ_1 \wedge xb$) =
(digraph-map (mk-map (mk-graph *iG*) *iM*) $\wedge$
($\forall$i $<$ |ig-es *iG*|. im-pred *iM* (im-succ *iM i*) = *i*)))
s

Some of the variable names are useful (as the VCG tries to preserve the names used in the program). However, it is customary to simplify the verification with a weak automatic proof method (for example clarsimp*) which is designed to "clarify" a subgoal. Doing so will lose any of the useful names inside the* case*-expressions:*

$\forall ia\ succ_1\ x1\ x1a\ x1b\ x2b.$
$succ_1 \longrightarrow x1 < |\text{ig-es}\ iG| \longrightarrow \neg\ x1a \Longrightarrow$
$\text{succ}_2\text{-inv}\ iG\ iM\ x1\ x1a\ x1b\ x2b \Longrightarrow$
$\text{succ}_1\text{-inv}\ iG\ iM\ ia\ succ_1 \Longrightarrow$
$\text{final}\ iG\ ia\ succ_1 \Longrightarrow$
$x1 \leq |\text{ig-es}\ iG| \Longrightarrow$
$\text{wf-digraph}\ (\text{mk-graph}\ iG) \Longrightarrow$
$(\text{bidirected-digraph}\ (\text{as-bidir}\ (\text{mk-map}\ (\text{mk-graph}\ iG)\ iM)) \wedge succ_1 \wedge x1a) =$
$(\text{digraph-map}\ (\text{mk-map}\ (\text{mk-graph}\ iG)\ iM) \wedge (\forall i < |\text{ig-es}\ iG|.\ \text{im-pred}\ iM\ (\text{im-succ}\ iM\ i) = i))$

7.3. Conclusion

The verification of the is-planar procedure completes the verified checker for the planarity test; provided a verified (or certifiying) algorithm to compute the number of components is available. The implementation consists of around 125 lines of proof text. Building on the theory about combinatorial planarity from Chapter 4, the verification takes about 1100 lines of proof text. In addition, 120 lines are necessary to provide the labeled rules for the implementation language. The verification was straightforward, requiring little inspiration, except for the invariants. Nevertheless, the correctness proofs for the procedures required a considerable amount of work and case labeling proved to be a valuable tool. While case labeling does not make the proof obligations simpler, it makes them easier to understand, closing an important gap to proof tools dedicated to program verification. Moreover, even for this small example, it greatly increases the readability and maintainability the proof.

8. Conclusion

Roughly speaking, this thesis covers two topics: formalization of graph theory and verification of checkers (for planarity certificates).

8.1. Results

The overarching theme of the first topic is the graph library I presented for reasoning about directed and, to some degree, undirected graphs. This library contains a formalization of many basic concepts and is flexible enough to be used with many different classes of digraphs. If not needed, this flexibility can be hidden, for example by using pair graphs, without losing the results of the general theory. Using this library I proved a characterization of Euler digraphs correct and formalized two characterizations of planarity: Kuratowski planarity and combinatorial planarity. I provided one half of the equivalence proof for these characterizations, that is comb-planar $G \implies$ kuratowski-planar G, and implemented a verified decision procedure for combinatorial planarity which is suitable for small graphs.

Independent of this graph library, I formalized a probabilistic proof of the Girth-Chromatic Number Theorem. Later, I transferred this result to the graph library using Isabelle's transfer mechanism. This required only shallow reasoning about the used constants and shows that results formalized with regard to other definitions of graphs can be embedded in my graph library with reasonable effort. The important point here is that the graph library is general enough to embed other graph representations as subtypes.

On the topic of verification, I verified checkers for both the planarity and the non-planarity certificates emitted by the certifying planarity test from the LEDA library. For the non-planarity checker, which is modeled after the implementation in LEDA, I showed that it accepts not only Kuratowski subgraphs but a slightly wider class of graphs which still guarantees non-planarity. This checker has been implemented in Simpl and in C. Both version have been verified. It turned out that abstraction of word arithmetic is crucial for the efficient verification of the C program. This prompted me to develop an abstraction framework which allows to separate reasoning about words from reasoning about the rest of the program and can additionally be use to insert ghost code.

The planarity checker has been implemented and verified in the abstract language of AutoCorres. This verification was used as a case study for my case labeling technique. Case labeling carries the program structure through the VCG over to the verification conditions, such that this structure can be exposed to the user with the help of Isabelle's cases mechanism. This makes proof more convenient and maintainable, as the user is relieved from manually matching verification conditions with fragments of the program. Beyond program verification, case labeling is applicable to many areas where proof obligations are structurally decomposed in a recursive fashion. Therefore, case labeling has the potential to make Isabelle's cases mechanism

more widely used, in particular for proof methods for which this was considered infeasible before One example for this are proofs for Isabelle's locale predicates.

8.2. Future Work

I conclude this thesis with a list of projects which seem to be a worthwhile continuation of this work.

Three Chapters of Graph Theory My graph library has been designed to be general, so it can useful for a wide range of graph theory. However, the selection of concepts and results I proved has been guided very much by my focus on planarity certificates. Similar to the work by Hölzl and Heller [37] for measure theory which provided the title so this paragraph, formalizing a few chapters of an introductory graph theory text book would give a wider base to this library.

Undirected Graphs, Hypergraphs, and More Some results, for example those of Chapter 4, can be smoothly formalized using digraphs-as-undirected-graphs. On the other hand, some of the proofs in Chapter 5 would probably have been easier with "real" undirected graphs. There are also proofs using mixed graphs [7] (i.e., graphs with both directed and undirected edges) and some of the formalized results can be generalized to hypergraphs. One structure which can support all these kinds of graphs natively is a triple of a set of vertices (of type β), a set of arcs (of type α), and a set of links (of type $\alpha \times (\beta \times \beta)$). Depending on the restrictions enforced for the set of links, such a structure would correspond to directed graphs, undirected graphs, directed or undirected hypergraphs, or mixed graphs. Properties like degree and walks can be defined in terms of links, uniformly for all this variants of graphs. It would be interesting to explore whether this even more general structure can be used for proofs about undirected graphs or digraphs without incurring too much overhead.

Equivalence between Kuratowski and Combinatorial Planarity I proved that combinatorial planarity implies Kuratowski planarity, but omitted the other direction. I expect that this proof would need a result equivalent to the Jordan Curve theorem, which is still missing from Isabelle.

Case Labeling For the case labeling, I presented a set of conditions which guarantee that a set of rules will generate a wellformed labeling. Labeling a set of rules according to these conditions is a very mechanical task. It is easy to imagine some lightweight annotations which supply just the names and from which the full labeling can be generated automatically.

Bibliography

[1] Behzad Akbarpour and Lawrence Paulson. "MetiTarski: An Automatic Theorem Prover for Real-Valued Special Functions". In: *Journal of Automated Reasoning* 44.3 (2010), pp. 175–205.

[2] Eyad Alkassar et al. "A Framework for the Verification of Certifying Computations". In: *Journal of Automated Reasoning* (2013), pp. 1–33.

[3] Noga Alon and Joel H. Spencer. *The Probabilistic Method*. 3rd ed. Wiley-Interscience, Aug. 2008.

[4] Philippe Audebaud and Christine Paulin-Mohring. "Proofs of Randomized Algorithms in Coq". In: *Science of Computer Programming* 74.8 (2009), pp. 568–589.

[5] R. J. R. Back. *Correctness Preserving Program Refinements: Proof Theory and Applications*. Mathematical Centre tracts. Mathematisch centrum, 1980.

[6] Clemens Ballarin. "Locales: A Module System for Mathematical Theories". In: *Journal of Automated Reasoning* (2013), pp. 1–31.

[7] Jørgen Bang-Jensen and Gregory Z. Gutin. *Digraphs: Theory, Algorithms and Applications*. 2nd ed. Springer, 2009.

[8] Gilles Barthe et al. "JACK — A Tool for Validation of Security and Behaviour of Java Applications". In: *Formal Methods for Components and Objects*. LNCS. 2007, pp. 152–174.

[9] Gertrud Bauer and Tobias Nipkow. "The 5 Colour Theorem in Isabelle/Isar". In: *Theorem Proving in Higher-Order Logics*. LNCS. 2002, pp. 67–82.

[10] Gertrud Bauer and Markus Wenzel. "Calculational Reasoning Revisited - an Isabelle/Isar Experience". In: *Theorem Proving in Higher-Order Logics*. LNCS. 2001.

[11] Johan F. A. K. van Benthem and Alice G. B. ter Meulen. *Generalized Quantifiers in Natural Language*. Walter de Gruyter, 1985.

[12] Manuel Blum and Sampath Kannan. "Designing Programs That Check Their Work". In: *Journal of the ACM* 42.1 (Jan. 1995), pp. 269–291.

[13] Béla Bollobás. *Random Graphs*. Academic Press, 1985.

[14] Nicolas Bourbaki. *General Topology (Part I)*. Addison-Wesley, 1966.

[15] R. W. Butler and J. A. Sjogren. *A PVS Graph Theory Library*. Tech. rep. NASA Langley, 1998.

[16] Ching-tsun Chou. "A Formal Theory of Undirected Graphs in Higher-Order Logic". In: *Theorem Proving in Higher-Order Logics*. LNCS. 1994, pp. 144–157.

[17] Ernie Cohen et al. “VCC: A Practical System for Verifying Concurrent C”. In: *Theorem Proving in Higher-Order Logics*. LNCS. Jan. 2009, pp. 23–42.

[18] Reinhard Diestel. *Graph Theory*. 4th ed. Vol. 173. GTM. Springer, 2010.

[19] Edsger Wybe Dijkstra. *Notes on Structured Programming*. Technological University Eindhoven Netherlands, 1970.

[20] Jean-François Dufourd. “An Intuitionistic Proof of a Discrete Form of the Jordan Curve Theorem Formalized in Coq with Combinatorial Hypermaps”. In: *Journal of Automated Reasoning* 43.1 (Mar. 2009), pp. 19–51.

[21] Jean Duprat. *A Coq Toolkit for Graph Theory*. Rapport de recherche 2001-15. LIP ENS Lyon, 2001.

[22] Noboru Endou, Keiko Narita, and Yasunari Shidama. “The Lebesgue Monotone Convergence Theorem”. In: *Formalized Mathematics* 16.2 (Jan. 2008). Formal proof development, pp. 167–175.

[23] P. Erdős and A. Hajnal. “On Chromatic Number of Graphs and Set-Systems”. In: *Acta Mathematica Academiae Scientiarum Hungaricae* 17.1-2 (Mar. 1966), pp. 61–99.

[24] P. Erdős and A. Rényi. “Asymmetric Graphs”. In: *Acta Mathematica Hungarica* 14.3 (1963), pp. 295–315.

[25] Paul Erdős. “Graph Theory and Probability”. In: *Canadian Journal of Mathematics* 11.11 (1959), pp. 34–38.

[26] Paul Erdős and Alfréd Rényi. “On Random Graphs I”. In: *Publicationes Mathematicae Debrecen* 6 (1959), pp. 290–297.

[27] Javier Esparza et al. “A Fully Verified Executable LTL Model Checker”. In: *Computer Aided Verification*. LNCS. July 2013, pp. 463–478.

[28] Holger Gast. “Semi-automatic Proofs about Object Graphs in Separation Logic”. In: *Electronic Communications of the EASST* 53 (Dec. 2012).

[29] Holger Gast. “Software Verification as Symbolic Debugging”. Unpublished. 2009.

[30] Georges Gonthier. “A computer-checked proof of the Four Colour Theorem”. 2005.

[31] Georges Gonthier. “Formal Proof–The Four-Color Theorem”. In: *Notices of the American Mathematical Society* 55.11 (Dec. 2008), pp. 1382–1393.

[32] Mike Gordon and Hélène Collavizza. “Forward with Hoare”. In: *Reflections on the Work of C.A.R. Hoare*. 2010, pp. 101–121.

[33] David Greenaway, June Andronick, and Gerwin Klein. “Bridging the Gap: Automatic Verified Abstraction of C”. In: *Interactive Theorem Proving*. LNCS. Jan. 2012, pp. 99–115.

[34] Thomas Hales. “A Proof of the Kepler Conjecture”. In: *Annals of Mathematics* 162.3 (Nov. 2005), pp. 1065–1185.

[35] Thomas Hales et al. “A Formal Proof of the Kepler Conjecture”. In: *arXiv:1501.02155 [cs, math]* (Jan. 2015).

[36] Frank Harary and Ronald C. Read. “Is the Null-Graph a Pointless Concept?” In: *Graphs and Combinatorics*. Vol. 406. Lecture Notes in Mathematics. 1974, pp. 37–44.

[37] Johannes Hölzl and Armin Heller. “Three Chapters of Measure Theory in Isabelle/HOL”. In: *Interactive Theorem Proving*. LNCS. 2011, pp. 135–151.

[38] Brian Huffman and Ondřej Kunčar. “Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL”. In: *Certified Programs and Proofs*. LNCS. Dec. 2013, pp. 131–146.

[39] Hunt, Warren A., Jr et al. “Meta Reasoning in ACL2”. In: *Theorem Proving in Higher-Order Logics*. 2005, pp. 163–178.

[40] Lars Hupel. *Interdisciplinary Project: Proofs about Random Graphs in Isabelle*. Tech. rep. 2012.

[41] Joe Hurd. “Formal Verification of Probabilistic Algorithms”. PhD thesis. University of Cambridge, 2002.

[42] Joe Hurd. “Verification of the Miller-Rabin Probabilistic Primality Test”. In: *Jounral of Logic and Algebraic Programming* 50.1–2 (2003), pp. 3–21.

[43] Casimir Kuratowski. “Sur le problème des courbes gauches en topologie”. In: *Fundamenta mathematicae* 15.1 (1930), pp. 271–283.

[44] Sergei K. Lando and Alexander K. Zvonkin. *Graphs on Surfaces and Their Applications*. Ed. by R. V. Gamkrelidze and V. A. Vassiliev. Vol. 141. Encyclopaedia of Mathematical Sciences. Springer Berlin Heidelberg, 2004.

[45] Gilbert Lee and Piotr Rudnicki. “Alternative Graph Structures”. In: *Formalized Mathematics* 13.2 (2005). Formal proof development, pp. 235–252.

[46] K. Rustan M. Leino, Todd Millstein, and James B. Saxe. “Generating error traces from verification-condition counterexamples”. In: *Science of Computer Programming*. FMCO 55.1–3 (Mar. 2005), pp. 209–226.

[47] David R Lester. “Topology in PVS: Continuous Mathematics with Applications”. In: *Automated Formal Methods*. AFM. 2007, pp. 11–20.

[48] László Lovász. “On Chromatic Number of Finite Set-Systems”. In: *Acta Mathematica Academiae Scientiarum Hungaricae* 19.1-2 (Mar. 1968), pp. 59–67.

[49] Daniel Matichuk, Makarius Wenzel, and Toby Murray. “An Isabelle Proof Method Language”. In: *Interactive Theorem Proving*. LNCS. July 2014, pp. 390–405.

[50] R. M. McConnell et al. “Certifying Algorithms”. In: *Computer Science Review* 5.2 (May 2011), pp. 119–161.

[51] Kurt Mehlhorn and Stefan Näher. “From Algorithms to Working Programs: On the Use of Program Checking in LEDA”. In: *Mathematical Foundations of Computer Science 1998*. LNCS. 1998, pp. 84–93.

[52] Kurt Mehlhorn and Stefan Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Nov. 1999.

[53] Farhad Mehta and Tobias Nipkow. “Proving Pointer Programs in Higher-Order Logic”. In: *Information and Computation* 199 (2005), pp. 200–227.

[54] Tarek Mhamdi, Osman Hasan, and Sofiène Tahar. "On the Formalization of the Lebesgue Integration Theory in HOL". In: *Interactive Theorem Proving*. LNCS. 2010, pp. 387–402.

[55] Bojan Mohar and Carsten Thomassen. *Graphs on Surfaces*. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, 2001.

[56] Yatsuka Nakamura and Piotr Rudnicki. "Euler Circuits and Paths". In: *Formalized Mathematics* 6.3 (1997). Formal proof development, pp. 417–425.

[57] Tobias Nipkow, Gertrud Bauer, and Paula Schultz. "Flyspeck I: Tame Graphs". In: *Automated Reasoning*. LNCS. 2006, pp. 21–35.

[58] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. LNCS. http://isabelle.in.tum.de/dist/Isabelle2015/doc/tutorial.pdf (Updated version). Springer, 2002.

[59] Benedikt Nordhoff and Peter Lammich. "Dijkstra's Shortest Path Algorithm". In: *Archive of Formal Proofs* (Jan. 2012). Formal proof development.

[60] Lars Noschinski. "A Graph Library for Isabelle". In: *Mathematics in Computer Science* 9.1 (June 2014), pp. 23–39.

[61] Lars Noschinski. "A Probabilistic Proof of the Girth-Chromatic Number Theorem". In: *Archive of Formal Proofs* (Feb. 2012). Formal proof development.

[62] Lars Noschinski. "Generating Cases from Labeled Subgoals". In: *Archive of Formal Proofs* (July 2015). Formal proof development.

[63] Lars Noschinski. "Graph Theory". In: *Archive of Formal Proofs* (Apr. 2013). Formal proof development.

[64] Lars Noschinski. "Planarity Certificates". In: *Archive of Formal Proofs* (Oct. 2015). Formal proof development.

[65] Lars Noschinski. "Proof Pearl: A Probabilistic Proof for the Girth-Chromatic Number Theorem". In: *Interactive Theorem Proving*. LNCS. 2012.

[66] Lars Noschinski. "Towards Structured Proofs for Program Verification (Ongoing Work)". Isabelle Workshop (Interactive Theorem Proving). 2014.

[67] Lars Noschinski, Christine Rizkallah, and Kurt Mehlhorn. "Verification of Certifying Computations through AutoCorres and Simpl". In: *NASA Formal Methods*. LNCS. Apr. 2014, pp. 46–61.

[68] Richard Rado. "Universal Graphs and Universal Functions". In: *Acta Arithmetica* 9 (1964), pp. 331–340.

[69] Christine Rizkallah. "An Axiomatic Characterization of the Single-Source Shortest Path Problem". In: *Archive of Formal Proofs* (May 2013). Formal proof development.

[70] Piotr Rudnicki and Lorna Stewart. "The Mycielskian of a Graph". In: *Formalized Mathematics* 19.1 (2011). Formal proof development, pp. 27–34.

[71] Norbert Schirmer. "Verification of Sequential Imperative Programs in Isabelle-HOL". PhD thesis. Technical University Munich, 2006.

[72] Christian Sternagel and René Thiemann. "Executable Transitive Closures of Finite Relations". In: *Archive of Formal Proofs* (Mar. 2011). Formal proof development.

[73] G.F. Sullivan and G.M. Masson. "Using Certification Trails to Achieve Software Fault Tolerance". In: 1990, pp. 423–431.

[74] Christoph Traut and Lars Noschinski. "Pattern-based Subterm Selection in Isabelle". In: *Isabelle Workshop (Interactive Theorem Proving)*. 2014.

[75] Dmitriy Traytel, Stefan Berghofer, and Tobias Nipkow. "Extending Hindley-Milner Type Inference with Coercive Structural Subtyping". In: *Programming Languages and Systems*. LNCS. 2011, pp. 89–104.

[76] Harvey Tuch, Gerwin Klein, and Michael Norrish. "Types, Bytes, and Separation Logic". In: *Principles of Programming Languages*. ACM SIGPLAN Notices. 2007, pp. 97–108.

[77] Lutz Volkmann. *Fundamente der Graphentheorie*. Springer, 1996.

[78] Markarius Wenzel. *The Isabelle/Isar Reference Manual*. May 2015. URL: http://isabelle.in.tum.de/dist/Isabelle2015/doc/isar-ref.pdf.

[79] Markus Wenzel. "Isabelle/Isar—a Versatile Environment for Human-Readable Formal Proof Documents". PhD thesis. Technische Universität München, 2002.

[80] John Wickerson, Mike Dodds, and Matthew Parkinson. "Ribbon Proofs for Separation Logic". In: *Programming Languages and Systems*. LNCS. Jan. 2013, pp. 189–208.

[81] Simon Winwood et al. "Mind the Gap: A Verification Framework for Low-Level C". In: *Theorem Proving in Higher-Order Logics*. LNCS. Aug. 2009, pp. 500–515.

[82] Niklaus Wirth. "Program Development by Stepwise Refinement". In: *Communications of the ACM* 14.4 (Apr. 1971), pp. 221–227.

[83] Wai Wong. "A Simple Graph Theory and its Application in Railway Signaling". In: *Theorem Proving in Higher-Order Logics*. 1991, pp. 395–409.

[84] Mitsuharu Yamamoto et al. "Formalization of planar graphs". In: *Higher Order Logic Theorem Proving and Its Applications*. LNCS. 1995, pp. 369–384.

A. A Language with State and Failure

Labeled rules for the language with state and failure used in Chapter 7.

$$\frac{\overbrace{\forall x.\ \{\!|R\ x|\!\}\ g\ x\ \{\!|Q|\!\}!}^{\text{ctxt } (o_1{+}1)\ (\langle\text{«bind»}, o_1{+}1, [r]\rangle_B{::}ct)\ o_2} \qquad \overbrace{\{\!|P|\!\}\ f\ \{\!|R|\!\}!}^{\text{ctxt } i\ ct\ o_1}}{\underbrace{\{\!|P|\!\}\ \text{bind}\ f\ (\lambda x.\ g\ x)\ \{\!|Q|\!\}!}_{\text{ctxt } i\ ct\ o_2}}$$

$$\frac{\overbrace{\{\!|P|\!\}\ f\ \{\!|Q|\!\}!}^{\text{ctxt } (i{+}1)\ ct\ o} \qquad \forall s.\ P's \implies \overbrace{P\ s}^{\text{vc } (\langle\text{«weaken»}, i, [s]\rangle_B{::}ct)}}{\underbrace{\{\!|P'|\!\}\ f\ \{\!|Q|\!\}!}_{\text{ctxt } i\ ct\ o}}$$

$$\frac{\forall y, z.\ \text{split}\ x\ (y, z) \implies \overbrace{\{\!|P\ y\ z|\!\}\ f\ y\ z\ \{\!|Q|\!\}!}^{\text{ctxt } i\ ct\ o}}{\underbrace{\{\!|\text{case-prod}\ P\ x|\!\}\ \text{case-prod}\ f\ x\ \{\!|Q|\!\}!}_{\text{ctxt } i\ ct\ o}} \qquad \frac{}{\underbrace{\{\!|\lambda s.\ P\ (f\ s)\ s|\!\}\ \text{gets}\ f\ \{\!|P|\!\}!}_{\text{ctxt } i\ ct\ (i{+}1)}}\ \text{Gets}_C$$

$$\frac{\overbrace{\{\!|L|\!\}\ l\ \{\!|Q|\!\}!}^{\text{ctxt } i\ (\langle\text{«then»}, i, []\rangle_B{::}ct)\ o_1} \qquad \overbrace{\{\!|R|\!\}\ r\ \{\!|Q|\!\}!}^{\text{ctxt } (o_1{+}1)\ (\langle\text{«else»}, o_2, []\rangle_B{::}ct)\ o_1}}{\underbrace{\{\!|\lambda s.\ (\text{if}\ c\ s\ \text{then}\ L\ s\ \text{else}\ R\ s)|\!\}\ \text{cond}\ c\ l\ r\ \{\!|Q|\!\}!}_{\text{ctxt } i\ ct\ o_2}} \qquad \frac{}{\underbrace{\{\!|P\ x|\!\}\ \text{return}\ x\ \{\!|P|\!\}!}_{\text{ctxt } i\ ct\ (i{+}1)}}$$

$$\frac{P_{\text{inv}} \qquad \forall x.\ \overbrace{\text{wf R}}^{\text{vc } (\langle\text{«wf»}, o, []\rangle_B{::}ct')} \qquad \forall x, s.\ [\![I\ x\ s; \neg c\ x\ s]\!] \implies \overbrace{Q\ x\ s}^{\text{vc } (\langle\text{«postcondition»}, (o{+}1), [s]\rangle_B{::}ct')}}{\underbrace{\{\!|I\ x|\!\}\ \text{while}\ c\ f\ x\ I\ R\ \{\!|Q|\!\}!}_{\text{ctxt } i\ ct\ (o{+}1)}}\ \text{While}_C$$

For the rule While_C, I used the following abbreviationsfollowing definitions:

$$ct' \coloneqq \langle\text{«while»}, i, [x]\rangle_B :: ct$$

$$P_{\text{inv}} \coloneqq \forall x, s.\ \overbrace{\{\!|\ \underbrace{I\ x}_{\text{tbind «linv» } i} \ \wedge\ \underbrace{c\ x}_{\text{tbind «lcond» } i} \ \wedge\ \underbrace{\lambda s'.\ s' = s}_{\text{tbind «lvar» } i}\ |\!\}\ f\ x\ \{\!|\lambda r'.\ \underbrace{I\ r'}_{\text{tbind «inv» } i} \wedge \underbrace{\lambda\_.\ (r', r) \in R}_{\text{tbind «var» } i}\ |\!\}}^{\text{ctxt } (i{+}1)\ (\langle\text{«inv»}, i, [s]\rangle_B{::}ct')\ o}$$