Jens Brandt

A Layered Approach to Polygon Processing for Safety-Critical Embedded Systems

22 June 2007

Vom Fachbereich Informatik der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades Doktor-Ingenieur (Dr.-Ing.)
genehmigte Dissertation

Datum der wissenschaftlichen Aussprache: 22. Juni 2007

Dekan: Prof. Dr. Reinhard Gotzhein
Vorsitzender der Prüfungskommission: Prof. Dr. Klaus Madlener
Erster Berichterstatter: Prof. Dr. Klaus Schneider
Zweiter Berichterstatter: Juniorprof. Dr. Georg Umlauf

D 386

Danksagung

Mein Dank gilt allen, die auf ihre Weise zum Gelingen meiner Promotion beigetragen haben.

Im Besonderen möchte ich mich bei Prof. Dr. Klaus Schneider bedanken, der mir die Möglichkeit eröffnet hat, diese Arbeit zu erstellen. Stetige Unterstützung, hilfreiche Diskussionen und fruchtbare Anregungen waren die ideale Begleitung meiner Promotion.

Für die Zweitbegutachtung bedanke ich mich bei Juniorprof. Dr. Georg Umlauf, der mir bei vielen Dingen eine zweite Sicht auf meine Arbeit liefern konnte. Für die Leitung der Promotionskommission bedanke ich mich bei Prof. Dr. Klaus Madlener, der mich mit Rat durch meine gesamte Studienzeit in Kaiserslautern begleitet hat. Ich bedanke mich ferner bei den Kollegen in meiner Arbeitsgruppe, die mir in Gesprächen einige neue Denkanstöße gaben.

Abschließend gilt meiner Famile mein besonderer Dank für die langjährige Unterstützung, ohne die ich nicht bis zu diesem Punkt gekommen wäre.

Kaiserslautern, 27. Juni 2007 *Jens Brandt*

Contents

Chapter 0

Zusammenfassung

Eingebettete Systeme sind in vielen Bereichen unseres heutigen Alltags mittlerweile unersetzlich. Ein sehr wichtiger Anwendungsbereich ist unter anderem die Automobilindustrie. Moderne Fahrzeuge sind nicht mehr nur das Werk ausgefeilter Maschinenbaukunst, sondern beinhalten auch komplexe Computersysteme, die manchmal aus mehr als 50 interagierenden Geräten bestehen.

Der Entwurf dieser Systeme wird durch neue Anwendungen und Anforderungen vor neue Herausforderungen gestellt. So besteht der Wunsch nach Systemen zur Situationsanalyse, die die Umgebung von Autos überwachen. Diese basieren im Allgemeinen auf Modellen und Algorithmen der algorithmischen Geometrie. Üblicherweise wird die Umgebung als Euklidische Ebene modelliert, in der Polygone verschiedene Objekte der Umgebung repräsentieren. Grundlegende Operationen der algorithmischen Geometrie können nun zur Darstellung des räumlichen Verhaltens verwendet werden. So kann zum Beispiel eine Kollision zweier Objekte durch einen nichtleeren Schnitt der korrespondieren Polygone festgestellt werden.

Übliche Implementierungen von Algorithmen der algorithmischen Geometrie sind nicht ohne weiteres für den Einsatz auf sicherheitskritischen eingebetteten Systemen geeignet. Zum einen fordern die hohen Sicherheitsanforderungen eine genaue Untersuchung der Korrektheit der Algorithmen. Zum anderen müssen nichtfunktionale Eigenschaften betreffend der begrenzten Ressourcen berücksichtigt werden.

Diese Dissertation schlägt ein schichtbasiertes Polygonverarbeitungssystem vor. Auf der Basis von rationalen Zahlen, wird ein Geometrie-Kernel formalisiert. Darauf aufbauend bilden geometrische Primitive eine weitere Abstraktionsebene, die für Plane-Sweep-Algorithmen und Polygone benutzt werden. Die Aufteilung in Schichten unterteilt das System nicht nur in handhabbare Teile, sondern ermöglicht es auch, auftretende Probleme auf den jeweils geeigneten Abstraktionsniveau zu betrachten. Dabei wird die Struktur konsequent in der Verifikation wie auch in der Implementierung der entwickelten Polygonverarbeitungsbibliothek eingesetzt.

Der Verwendung in sicherheitskritischen Systemen wird durch ein Verifikationsframework Rechnung getragen. Die innerhalb des interaktiven Theorembeweisers HOL entwickelten Theorien stellen verschiedene Abtraktionsebenen dar und bieten dafür jeweils eine Reihe von Hilfsmitteln an, die zum Spezifizieren und Verifizieren von Polygonverarbeitunsalgorithmen genutzt werden können: Neben einem formalisierten Geometrie-Kernels sind häufig im Anwendungsgebiet genutzte Operationen definiert und Eigenschaften dafür bewiesen, auf denen bei der Verifikation von Algorithmen aufgebaut werden kann.

Die Implementierung profitiert ebenfalls von der Schichtenarchitektur. So orientiert sie sich an der Formalisierung und ähnelt sehr den entwickelten Theorien: Auf dem verifizierten Geometrie-

Kernel sind geometrische Primitiven implementiert, die die Basisbausteine für die Softwarebibliothek bilden.

Um die entwickelte Bibliothek in eingebetteten Systemen nutzen zu können, sind zusätzliche nichtfunktionale Anforderungen in der verschiedenen Schichten des Systems berücksichtigt. Konservative Heuristiken zur Vereinfachungen des Modell sind ein integrales Bestandteil der Bibliothek: Sie begrenzen die Komplexität der Probleme und damit der Laufzeit, so dass die Algorithmen auf eingebetteten Systemen mit eingeschränkten Ressourcen lauffähig sind. Besonderer Wert wurde darauf gelegt, dass alle Algorithmen so einfach wie nur möglich sind.

Das Ergebnis ist eine verlässliche Softwarebibliothek zur Polygonverarbeitung für eingebettete Systeme. Ein Prototyp wurde in der Programmiersprache C implementiert und im Kontext eines Situationsanalysesystems für autonome mobile Roboter evaluiert.

Chapter 0

Abstract

Embedded systems have become ubiquitous in everyday life, and especially in the automotive industry. Modern cars are not only the result of mechanical engineering, but they also contain complex computer systems, which sometimes consist of more than 50 interacting embedded devices.

New applications challenge their design by introducing a new class of problems that are based on a detailed analysis of the environmental situation. This situation analysis relies on models and algorithms of the domain of computational geometry. The basic model is usually an Euclidean plane, which contains polygons to represent the objects of the environment. Primitive operations of computational geometry are used to model the spatial behaviour: For example, checking whether two objects collide is equivalent to checking whether their corresponding geometric objects have a non-empty intersection.

Usual implementations of computational geometry algorithms cannot be directly used for safety-critical systems. First, a strict analysis of their correctness is indispensable and second, non-functional requirements with respect to the limited resources must be considered.

This thesis proposes a layered approach to a polygon-processing system. On top of rational numbers, a geometry kernel is formalised at first. Subsequently, geometric primitives form a second layer of abstraction that is used for plane sweep and polygon algorithms. These layers do not only divide the whole system into manageable parts but make it possible to model problems and reason about them at the appropriate level of abstraction. This structure is used for the verification as well as the implementation of the developed polygon-processing library.

The application area of safety-critical systems is paid attention by the development of a verification framework based on the interactive theorem prover HOL. Its main goal is to provide different levels of abstraction for verification tasks and to give a tool set that can be generally applied for specifying and verifying polygon-processing algorithms: Besides a formalised geometry kernel, other common geometric operations of the application domain are defined, and various theorems for them are proven so that future work can be built on this basis.

The implementation also benefits from the layered approach. Its structure is closely related to the formalisation and resembles the developed theories: On top of the verified geometry kernel, geometric primitives are implemented that are used for the software library.

To apply the developed polygon-processing library in embedded systems, additional nonfunctional requirements are implemented by the different layers of the system: Conservative simplification heuristics to limit the problem sizes and thus the execution time, while retaining the correctness, are an inherent part of the library. Moreover, in contrast to general-purpose implementations, all algorithms are designed to be as simple as possible to cope with the limited resources of embedded systems.

The result is a dependable polygon-processing software library for safety-critical embedded systems. A prototype implemented in the programming language C is evaluated with the help of a situation-analysis system for autonomous mobile robots.

Chapter 1

Introduction

1.1 Motivation

Embedded systems have become ubiquitous in everyday life. Albeit hidden from the user, most electronic devices contain small controllers with a remarkable amount of software, which primarily contributes to the functionality, comfort and safety. In recent years, these systems are more and more responsible for the economic success of industrial products.

A very important application domain of embedded systems is the automotive industry. Modern cars are not only the result of mechanical engineering, but they also contain complex computer systems, which sometimes consist of more than 50 interacting embedded devices. On one side, these systems are responsible for the majority of innovations, and sometimes car models only differ in features realised by them. On the other side, with their increasing complexity they are meanwhile responsible for the majority of break-downs.

So far, embedded systems have been used to monitor and optimise the function of internal components of cars like fuel injection and anti-lock braking systems. With the improvement of radar and optical sensors, which are already available in many cars, new applications came in the focus of development. Since cars are now aware of the objects in their environments, future systems will be able to analyse the current environmental situation in order to assist the driver and even to actively take control over some of his tasks: As already reality in avionics, these systems can undertake routine tasks or supervise human actions to avoid serious accidents.

These new applications challenge the design of embedded systems by introducing a new class of problems that are based on a detailed analysis of the environmental situation. This situation analysis relies on models and algorithms of the domain of computational geometry. In industrial applications, there is no experience in such problems — quite in contrast to robotics, where these problems have been considered from the beginnings.

For example, dynamic motion planning or collision avoidance for mobile autonomous robots are similar problems, which essentially rely on geometric computations. The basic model is usually an Euclidean plane, which contains polygons to represent the objects of the environment. Primitive operations of computational geometry are used to model the spatial behaviour: For example, checking whether two objects collide is equivalent to checking whether their corresponding geometric objects have a non-empty intersection.

The systems previously developed in robotics demonstrate the concepts, but they can be merely seen as prototypes, which lack essential properties for an integration into industrial systems: First, industrial systems in the automotive domain are highly safety-critical, hence errors are intolerable in real-world applications. Second, as industrial applications are always subject to economic constraints, the algorithms must be run on less powerful computing devices, which involves optimisation and abstraction problems.

1.2 Contribution

This thesis proposes a layered approach to a polygon-processing system. These layers do not only divide the whole system into manageable parts but make it possible to model problems and reason about them at the appropriate level of abstraction. This structure is used for the verification as well as the implementation of the developed library:

First, the thesis contributes a framework[1] that can be generally used to specify and verify polygon-processing algorithms. Its main goal is to provide different levels of abstraction for verification tasks and to give a tool set that can be generally applied for specifying and verifying polygon-processing algorithms. Basic geometric operations that are common elements of algorithms are formalised, and various theorems for them are proven so that future work can be built on this basis. On top of the analytic geometry basics, primitives are the first abstraction layer, followed by common data and control structures for polygon-processing algorithms.

Second, the layers are used to modify computational geometry algorithms that are the basis of motion planning and collision avoidance procedures so that they can be used in safety-critical embedded devices. On top of a verified geometry kernel, a dependable software library for polygon processing is built from the same layers, which are augmented by additional nonfunctional tasks. Heuristics to limit the problem sizes and thus the execution time are an inherent part of the library. Moreover, in contrast to general-purpose implementations as provided by common libraries like CGAL [20] or LEDA [98], all algorithms are designed to be as simple as possible to cope with the limited resources of embedded systems.

The results are both a general reusable structure of a polygon-processing framework and a concrete implementation of a polygon-processing software library for safety-critical embedded systems. The prototype was implemented in C and was evaluated with the help of a situation-analysis system for autonomous mobile robots.

The rest of the introduction gives an overview of the two domains of computer science that form the basis of the work: computational geometry and embedded systems. In addition, each section discusses the requirements of the developed software library with respect to the characteristics of the respective domain.

1.3 Related Areas

1.3.1 Computational Geometry

Problem Classes

Computational geometry [36] denotes the field of computer science that deals with problems stated in terms of geometry. Basically, it considers three problem classes.

[1] available through the author's webpage

The first one is the classification problem: a situation is given, and the algorithm has to classify the problem up to a predefined number of cases. For example, the *point-in-polygon* problem is to decide whether a point is inside or outside a given polygon. Or, given two line segments, an algorithm determines whether there is a *line intersection* or not.

These algorithms are usually the basis of the second class of problems, where some geometric objects are given as inputs and the output should be some other geometric object that is in a certain relation to the given ones. The *convex hull* problem consists in finding the smallest polygon that contains all the points of other polygons. As another example, the *triangulation* of a polygon is a partition of its interior into triangles by connecting the vertices.

The third class of problems focuses on data structures. A given input is organised in a way that some kind of query can be processed efficiently. The problem may involve other operations, like the modification, insertion or removal of some input objects. For example, the *range searching* problem processes a set of points, in order to efficiently count the number of points inside a query region. For the *nearest neighbour* problem, a set of points is preprocessed to efficiently find which point is closest to a query point.
The algorithms discussed in this thesis basically belong to the first two groups of algorithms.

Applications

Since its beginning in the 1970s, a lot of research was done in the area of computational geometry. A lot of algorithms and data structures that have been developed are applied in various domains, among them the following ones [36]:

- *Computer graphics:* Generally, computer graphics [49] deal with bringing an image of a modeled scene on a computer device, such as a screen or a printer. The model is usually either two or three dimensional and may contain thousands or millions of vertices, edges and faces. An exemplary problem for a correct display is the removal of hidden parts of the model, which are invisible from a particular viewpoint. More complex problems deal with light sources, textures or the computation of shadows. All applications of the domain require the efficient implementation of geometric algorithms.
- *Geographic information systems:* These systems [17, 31] consist of databases that store geographical data, e. g. the location of cities or the shape of roads and countries. The typical problem is to extract information about stored objects and to analyse the relation between different types of data. The main problem is the design of efficient data structures to store and later retrieve geographical information. Another problem is the combination of maps, which is usually known as the map overlay problem, which will be discussed in detail in Chapter 6.
- *CAD:* In our days, a lot of products are designed with the help of a computer [48, 76, 103]; examples are circuit boards, machine parts, furniture or complete buildings. Each of these products can be seen as a geometric entity, which has certain properties. The model can be used for simulations and supports engineers to design products that can be assembled in an easy way.
- *Robotics:* Robots [55, 79, 88, 96, 117] that are aware of their environment usually model it with objects in the Euclidean plane or space. An example are industrial robots that use an arm for their operations. Questions whether (or how) a certain point can be reached with the arm can be reduced to geometric problems. If the robot itself is a mobile object, interactions with

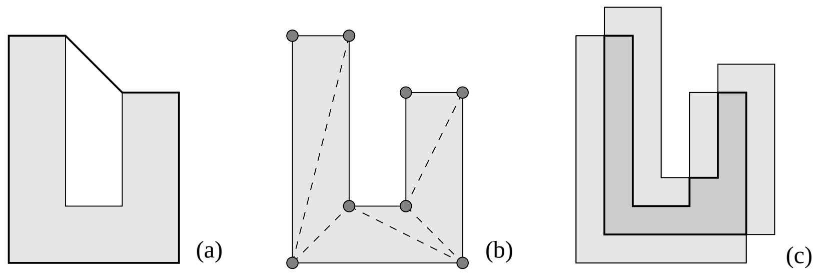

Fig. 1. Polygon Processing Algorithms

the environment are even more interesting. Typical problems are motion planning or collision detection, which again have a geometrical basis.

As already stated previously, the last application area is considered here.

Polygon Processing

In this thesis, the considerations are restricted to two-dimensional, linear objects like lines, segments, and polygons, as they are the basis for the target applications. Related algorithms (see Figure 1), e. g. to determine the convex hull (a), to triangulate a polygon (b), and to compute set operations on a given set of polygons (c), are the the main object of study. In principle, the limitation to two dimensions is not severe, since all considerations could be generalised to higher dimensions and other geometric objects.

1.3.2 Safety-Critical Embedded Systems

In computational geometry, much effort has been spent in developing algorithms to efficiently solve the fundamental problems of this domain. Seemingly every aspect of these problems has been studied in detail and great efforts have been made to transform the initial ideas into robust and efficient implementations as available in the LEDA system [98] or in the CGAL library [20]. In particular, there are solutions to the problems given in Figure 1 [36]. However, these implementations cannot be directly deployed in safety-critical embedded systems, since additional non-functional requirements have to be considered. In the following, these aspects are discussed, which are related to the domain of safety-critical embedded systems [95, 129].

Industrial Context

Embedded systems are found in many modern industrial products. Hidden from the actual user interface, they have specialised components that have been always designed for a particular application. They are usually heterogeneous containing software as well as hardware parts (see Figure 2). Latter ones may be built from standard architectures, but custom components are quite common if high-performance computing is required, e. g. for digital signal processing. On the other side, their software has to be very efficient to keep hardware costs as low as possible.

Custom hardware and the integration in a surrounding system make modifications to an embedded system a very difficult task so that errors usually have severe consequences on the economic success of a system.

Correctness

Industrial embedded systems are often found in safety-critical areas, and thus, they must meet high quality standards like functional correctness and dependability in a general sense. The stakes are exceedingly high, and design errors potentially lead to severe damages including even the loss of human lives. So, they must be developed with great care. It is therefore problematic that computational geometry algorithms usually impose many tricky cases and, with them, unforeseen problems that challenge the design of robust embedded systems. Their complexity is commonly underestimated due to their seemingly intuitive nature. As a consequence, the difficulty of geometric algorithms is often underestimated and the algorithms are often only loosely described by means of pictures. However, at a second glance, even simple definitions turn out to be much more complicated than expected: For example, what is the intersection point of two lines, if both lines are identical? For this reason, most algorithms sketched in literature only work under certain preconditions like ‘all points are pairwise distinct’ or ‘no three points are collinear’. Finding consistent definitions that also hold for degenerate input data is surprisingly difficult, which makes it inherently difficult to develop a polygon-processing library for safety-critical embedded systems.

In the area of embedded systems, formal verification is already routinely performed as a complementary technique to testing and simulation in order to ensure the correctness of the system. Only formal methods provide languages whose meaning is clear enough to reason about required definitions and algorithms without ambiguity. In order to also apply verification to new applications that have their origins in computational geometry, critical parts must be formalised so that relevant specifications can be checked with theorem provers like the HOL [60] system. This rigorously formal approach ensures that subtle cases will not be overseen and that all definitions are consistently used.

The verification guarantees the correctness of the algorithmic part of the system. However, implementation details, in particular the sensor part, remain unverified.

Simplification

Many embedded systems belong to the class of reactive systems, which are characterised by their non-terminating, continuous interaction with their environment. In contrast to interactive systems, the interaction points with the environment are not determined by the system. Instead, they must

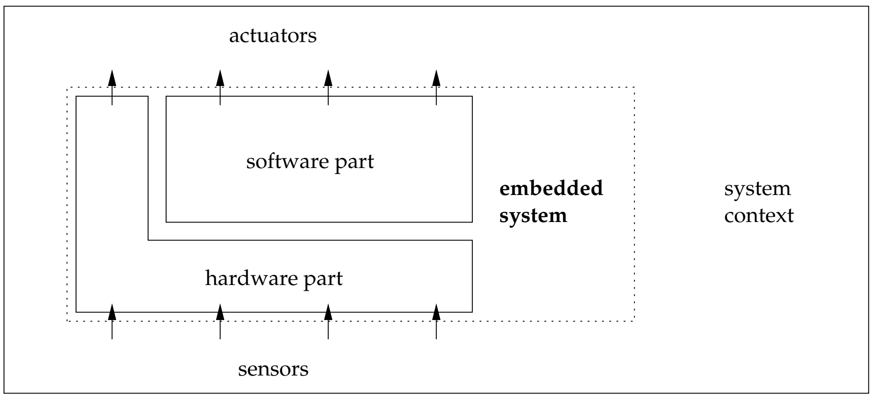

Fig. 2. Schematic Structure of an Embedded System

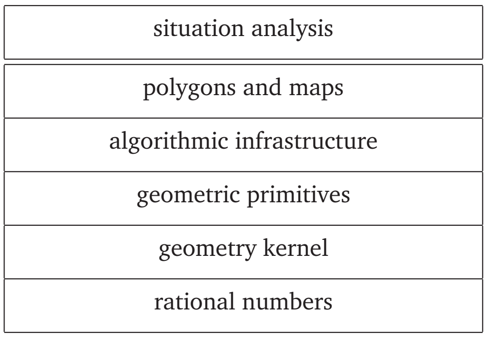

Fig. 3. Structure of the Polygon-Processing Library

react to the environment within a fixed response time. Systems that operate under these conditions are usually called real-time systems.

Moreover, embedded systems are subject to economic constraints. Computation and memory resources are dimensioned to an absolutely required minimum. hence, all algorithms must be implemented very efficiently. Moreover, in order to cope with hard real-time constraints, the resource requirements must be limited. Conservative simplification techniques, which do not impair the safety of the overall system, are a key component to meet the real-time requirements.

Rounding

The limited precision of the arithmetic operations of microprocessors leads to a loss of information. For safety-critical systems, this loss of information may be fatal, since it may lead to inconsistent situations that may consecutively lead to unexpected reactions of the system. In particular, using the usual rounding methods as provided by the IEEE-754 standard for floating point numbers may even lead to completely wrong results in geometric computations.

For this reason, some researchers propose the use of arbitrarily precise arithmetic or interval arithmetic [63, 113, 123] to circumvent these problems. As these solutions are too complex for small embedded devices, there is a need for explicit rounding functions that approximate the results in a conservative manner according to the specific needs of the particular algorithms.

1.4 Structure of the Thesis

The structure of the thesis follows the structure of the developed system (see Figure 3). Each chapter describes one of its layers — beginning with the basis of rational numbers and ending with polygons and maps. Additionally, a system for analysing the environmental situation that uses the polygon-processing library will be presented.

Chapter 2 starts with the verification foundations. It introduces the HOL system, an interactive theorem prover and proof checker that has been used for all formalisations of this thesis. It features an extensive library of mathematical theories and proof tools, which is extended by a theory of rational numbers.

On top of this basis, a geometry kernel is developed in Chapter 3. Arbitrary-precision rational numbers as considered before are used as the underlying arithmetic. All operations are formalised within a theory of two-dimensional analytic geometry and are implemented accordingly.

So, reasoning about the correctness of geometric computations becomes possible. Arbitrary precision rational numbers are provided to the polygon-processing C library by the GNU Multiprecision package [54].

Chapter 4 establishes an abstraction layer formed by geometric primitives. Hiding the details of analytic geometry, higher-level programming and verification becomes possible. The definition of these primitives is not straightforward: even simple cases pose tricky problems: For instance, what is the interior of a general polygon? The theorem prover HOL is used to reason about the formalisations and perform a comprehensive analysis in order to guarantee that all possible cases are handled consistently and that computational geometry algorithms work as expected in all possible cases. Three-valued logic is used from this layer upwards for concise definitions and algorithms.

A fundamental approach that is used in many computational geometry algorithms is the subject of Chapter 5: It presents the class of plane sweep algorithms. Its general paradigm is integrated in the polygon-processing library by providing functions and data structures. Moreover, the plane sweep is analogously integrated in the verification framework.

Chapter 6 introduces maps as data structures to describe areas in the plane. They form the basis for polygon processing, the application considered in this thesis. Especially, the definitions of the interior and exterior of regions are crucial points for the verification of map overlay algorithms. A new algorithm for this problem is given, which respects the non-functional requirements of the developed library.

On top of the polygon-processing library Chapter 7 constructs a situation-analysis systems for autonomous mobile robots. It illustrates some new applications that become available with the developed library. Furthermore, its efficiency is compared to other libraries that meet the same (functional) requirements.

Finally, Chapter 8 sums up the results and draws some conclusions.

Chapter 2

Formal Foundations

As already stated in the introduction, the verification of the polygon-processing library is a very important task of this thesis. This chapter lays the necessary foundations for this work.

In the following, an overview of the HOL System is given in Section 2.1. Its logical foundations are presented, and it is explained how the system can be used to make definitions and to reason about them. In Section 2.2, a theory of rational numbers and related tools are developed. First, this illustrates the usage of the theorem prover, and second, the rational numbers theory and library are used as a basis for all layers of the polygon-processing library. Although not all details are important for the understanding of the rest of the thesis, the description of the theory gives a good impression about the work with a theorem prover.

2.1 The HOL System

The HOL system [60, 77] is an interactive theorem prover for higher order logic with polymorphic types. HOL has been developed by various researchers around the world for more than twenty years. The system provides many data types, mathematical theories and proof tools, which can be extended by users to adapt the system to personal needs. HOL has been used in many areas, including hardware design and verification, reasoning about security, semantics of programming languages, reasoning about real-time systems and software verification. For an (outdated by nevertheless excellent) overview, see the paper by Kalvala [83].

The roots of the HOL family of theorem provers reach back to the LCF (Logic of Computable Functions) system [61] developed by Milner, Gordon and others in Stanford and Edinburgh in the early 70s. Like many other theorem provers (Coq[34], Isabelle[81, 108], NuPRL[109]), it still follows the philosophy and main design principles of LCF, e. g. the protection of theorems with an abstract data type. HOL evolved from LCF when Gordon developed a proof system for hardware verification [19, 58]. Higher order logic is then a natural choice for description: Signals are mapped to functions from a time data type to Booleans, and hardware components are just relations from signals to signals.

2.1.1 Overview

The HOL logic is based on Church's theory of simple types [30], which is extended by two significant aspects: First, polymorphic types developed by Milner for the PCF logic PPλ [61] are added. Second, it contains formal rules of definition for new constants and new types.

In HOL, theorems are of the form

$$\texttt{THEOREM\_NAME} \quad t_1, \ldots, t_n \vdash t$$

where the assumptions $t_1, \ldots, t_n$ and the conclusion t are Boolean terms. Such a theorem asserts that if its assumptions are true, then also its conclusion is true. Hence, such a theorem is equivalent to the validity of the formula $t_1 \implies \ldots \implies t_n \implies t$, which will be used in the following. So, theorems are simply written as

$$\texttt{THEOREM\_NAME} \quad \vdash t_1 \implies \ldots \implies t_n \implies t$$

HOL's calculus consists of five axioms and eight primitive inference rules (see Section 2.1.2): This deductive system is shown to be sound, and all extensions of theories by definitions of types and constants are conservative, i. e. they preserve the consistency of the theory, since their existence has to be proven in advance. Thus, all HOL theories are guaranteed to be consistent by construction.

HOL is implemented in Moscow ML [104], a light-weight implementation of the strict functional programming language ML [69, 110]. HOL's open structure allows the users to implement special proof rules and tactics in ML on top of HOL's functions for their particular application[1]. Types and terms of higher order logic are thereby implemented as data types in ML together with corresponding constructors. Theorems are represented by an ML data type `thm`. As there is no primitive constructor for this data type, the system is protected from creation of arbitrary theorems. The only way to create them is by presenting the system a proof, which can be achieved by forward (see Section 2.1.2) and backward reasoning (see Section 2.1.3).

As higher order logic is a very powerful logic, automated theorem proving is only possible in a limited way. For the proof construction the system either requires hints from a human user in an interactive session, or tactics that have been previously defined by the user. HOL's meta language can be used for programming and extending the theorem prover: New and more powerful inference rules can be obtained by combining simpler rules, and definitions and theorems can be aggregated to form new theories for later use. In this way, the metalanguage makes efficient proofs consisting of millions of derivation steps possible. For many types (including the naturals, integers and reals), HOL has so-called simplifiers, which feature contextual rewriting with conditions and embedded decision procedures. Moreover, resolution and model elimination for first-order reasoning are integrated.

The HOL system comes with dozens of theories providing hundreds of theorems, inference rules, tactics and other useful functions. As each theorem is derived from the core of the system, i.e. the axioms and the primitive inference rules, the HOL system ensures that only consistent theories are created. In this way, a large system can be built on top of a small trusted kernel.

2.1.2 System Kernel

As a theorem prover is intended to check other systems, it must be very reliable. The HOL system achieves this by restricting to a very small kernel. Naive and pure implementations of the central part fit on a few pages and can be easily inspected by hand. All things outside this kernel are designed to be checked by the kernel, so that the correctness of the system can never be affected

[1] This was the original purpose of the ML programming language, a $\underline{m}$eta $\underline{l}$anguage for the LCF theorem prover [61].

by programming errors. The following paragraphs present the core types and key concepts of the HOL system, which are implemented in the system kernel. For a detailed description, especially the semantics, refer to [60] or the HOL reference [77].

Types

The HOL logic is based on Church's theory of simple types [30]. Its syntax contains categories that denote certain sets and elements of them. These are taken from the universe $\mathcal{U}$, which is defined as follows [60, 77]:

Definition 1 (Universe). *The universe $\mathcal{U}$ is a fixed set of sets that has the following properties:*

- *Each element $X \in \mathcal{U}$ of the universe is non-empty.*
- *If a set in the universe $X \in \mathcal{U}$ has a non-empty subset $Y \subseteq X$, then the subset is also in the universe $Y \in \mathcal{U}$.*
- *If $X \in \mathcal{U}$ and $Y \in \mathcal{U}$, then the Cartesian product is also in the universe $X \times Y \in \mathcal{U}$. The set $X \times Y$ consists of the ordered pairs (x, y), where $x \in X$ and $y \in Y$.*
- *If $X \in \mathcal{U}$, then the powerset $\wp(X) = \{Y : Y \subseteq X\}$ is also in the universe $\mathcal{U}$.*
- *The universe $\mathcal{U}$ contains a distinguished infinite set I.*
- *There is a distinguished element $ch \in \Pi_{X \in \mathcal{U}} X$. The elements of the product $\Pi_{X \in \mathcal{U}} X$ are dependently typed functions: Thus, for all $X \in \mathcal{U}$, X is non-empty by the first property and $ch(X) \in X$ is the witness.*

From these properties, it follows that the universe $\mathcal{U}$ contains the set of functions $X \rightarrow Y \in \mathcal{U}$ (if $X \in \mathcal{U}$ and $Y \in \mathcal{U}$) and a distinguished two-element set $\mathbb{B}$.

Definition 2 (HOL Types). *Types of the* HOL *logic are expressions that determine elements of the universe $\mathcal{U}$. Basically, there are four different kinds:*

- Atomic types *denote fixed sets of the universe $\mathcal{U}$. In each theory, a particular collection of atomic types is determined. For example, the standard atomic type* `bool` *denotes the two-element set $\mathbb{B}$.*
- Compound types *are built from other types $\sigma_1, \ldots \sigma_n$. For a given operation, $(\sigma_1, \ldots, \sigma_n)op$ is a type denoting the set resulting from the application of op to the sets $\sigma_1, \ldots, \sigma_n$. For example,* prod *is a type operator of arity 2 that denotes the Cartesian product operation.*
- Function types *are built from two other types σ_1 and σ_2: $\sigma_1 \rightarrow \sigma_2$ is the function type with domain σ_1 and range σ_2. It denotes the set of total functions from the set denoted by its domain to the set denoted by its range. In principle, $\rightarrow$ is simply a distinguished type operator of arity 2. However, it is singled out in the definition of* HOL *types, since it always denotes the same operation in any model of the* HOL *theory.*
- Type variables*: They stand for arbitrary sets in the universe.*

Terms

Terms in the HOL logic are expressions that denote elements of the sets denoted by types. Thus, each term t is associated with a unique type σ that is expressed by t_σ. As the definition of types in HOL is relative to a particular type structure Ω, the formal definition of terms is relative to a given collection of typed constants over Ω. A signature over Ω is just a set Σ_Ω of such constants.

	HOL notation	standard notation	description
Truth	`T`	T	*true*
Falsity	`F`	F	*false*
Negation	`~`t	$\neg t$	*not* t
Disjunction	t_1`\/`t_2	$t_1 \vee t_2$	t_1 *or* t_2
Conjunction	t_1`/\`t_2	$t_1 \wedge t_2$	t_1 *and* t_2
Implication	t_1`==>`t_2	$t_1 \implies t_2$	t_1 *implies* t_2
Equality	t_1`=`t_2	$t_1 = t_2$	t_1 *equals* t_2
$\forall$-quantification	`!`x`.`t	$\forall x.t$	*for all* $x : t$
$\exists$-quantification	`?`x`.`t	$\exists x.t$	*for some* $x : t$
ε-term	`@`x`.`t	$\varepsilon x.t$	*an* x *such that* t

Fig. 4. Logical HOL Constants [60]

Definition 3 (HOL Terms). *The set of terms over Σ_Ω is defined to be the smallest set closed under the formation rules:*

- Constants*: A constant c_σ over Ω is a pair (c, σ) where $c \in$ Names and $\sigma \in Types_\omega$.*
  (Assume that an infinite set Names *of names is given.)*
- Variables*: If $x \in$ Names and $\sigma \in$ Types$_\Omega$ then the variable* var x_σ *is a term over Σ_Ω.*
- Lambda-abstractions*: The lambda-abstraction (which denotes a function) $(\lambda x_{\sigma_1}.t_{\sigma_2})_{\sigma_1 \to \sigma_2}$ is a term if* var $x_{\sigma_1} \in$ Terms$_{\Sigma_\Omega}$ *and $t_{\sigma_2} \in$ Terms$_{\Sigma_\Omega}$.*
- Function applications*: If $t_\sigma \in$ Terms$_{\Sigma_\Omega}$ and $t'_{\sigma'} \in$ Terms$_{\Sigma_\Omega}$ then $(t_{\sigma' \to \sigma}\ t'_{\sigma'})_\sigma \in$ Terms. An application $t\,t'$ denotes the result of applying the function denoted by t to the value denoted by t'.*

Standard structures

A standard type structure Ω contains the atomic types $\mathbb{B}$ of Boolean values and P of individuals. Logical formulas are then identified with terms of type $\mathbb{B}$. In addition, for being standard a signature must contain various logical constants with interpretation given by a standard model M:

- *Implication*: $\implies$ of type $(\mathbb{B} \to \mathbb{B} \to \mathbb{B})$ represents the implication. The intended interpretation $M(\Rightarrow, \mathbb{B} \to \mathbb{B} \to \mathbb{B})$ is as follows: $b \implies b'$ is 0, if $b = 1$ and $b' = 0$; otherwise $b \implies b'$ is 1.
- *Equality*: $=$ of type $(\alpha \to \alpha \to \mathbb{B})$ denotes equality on the set denoted by α. The intended interpretation $M(=, \alpha \to \alpha \to \mathbb{B})$ is as follows: $x = x'$ is 1 if and only if x is equal to x'; otherwise it is 0.
- *Choice function*: ε of type $((\alpha \to \mathbb{B}) \to \alpha)$ is Hilbrt's choice function. The intended interpretation $M(\varepsilon, (\alpha \to \mathbb{B}) \to \alpha)$ is as follows: εP selects an element x of type α such that $P(x)$ holds. If no such element exists, εP denotes an arbitrary element of type α (see Definition 1).

The minimal theory of HOL consists only of these standard items without any axioms. In the next step, the standard set of logical constants (see Figure 4) is constructed with the help of these basic primitives. There are several possibilities, e. g. the following one:

`EXISTS` $\vdash_{\mathrm{def}} \exists = \lambda P.P(\varepsilon P)$
`TRUTH` $\vdash_{\mathrm{def}} \mathsf{T} = ((\lambda x.x) = (\lambda x.x))$
`FORALL` $\vdash_{\mathrm{def}} \forall = \lambda P.(P = (\lambda x.\mathsf{T}))$
`FALSITY` $\vdash_{\mathrm{def}} \mathsf{F} = \forall x.x$
`NEGATION` $\vdash_{\mathrm{def}} \neg = \lambda x.x \implies \mathsf{F}$
`DISJUNCTION` $\vdash_{\mathrm{def}} \vee = \lambda(x,y).\neg x \implies y$
`CONJUNCTION` $\vdash_{\mathrm{def}} \wedge = \lambda(x,y).\neg(\neg x \vee \neg y)$

Deductive System

So far, only types and terms have been defined. For the creation of proofs, the deductive system has to be added. This system basically defines which formulas are theorems. This is either done explicitly by giving a term, which is then called an axiom. Or, the theorem can be derived by other theorems and inference rules. The deductive system of HOL is shown to be sound: Theories constructed on top of the initial HOL theory are guaranteed to be consistent.

Primitive Inference Rules

Inference rules are constructed by the following eight primitive rules [77]:

- *Assumption Introduction*

$$\texttt{ASM\_INTRO} \quad \frac{}{t \vdash t}$$

- *Reflexivity*

$$\texttt{REFLEX} \quad \frac{}{\vdash t = t}$$

- *Beta conversion*: Let $t_1[t_2/x]$ be the result of substituting t_2 for x in t_1, with suitable renaming of variables to prevent free variables in t_2 becoming bound after substitution.

$$\texttt{BETA\_CONV} \quad \frac{}{\vdash (\lambda x.t_1)t_2 = t_1[t_2/x]}$$

- *Substitution*: Let $t[t_1, \ldots, t_n]$ denote a term t with some free occurrences of subterms $t_1, \ldots, t_n$ singled out, and let $t[t'_1, \ldots, t'_n]$ denote the result of replacing each selected occurrence of t_i by t'_i (for $1 \leq i \leq n$), with suitable renaming of variables to prevent free variables in t'_i becoming bound after substitution.

$$\texttt{SUBST} \quad \frac{\Gamma_1 \vdash t_1 = t'_1 \quad \cdots \quad \Gamma_n \vdash t_n = t'_n \qquad \Gamma \vdash t[t_1, \ldots, t_n]}{\Gamma_1 \cup \cdots \cup \Gamma_n \cup \Gamma \vdash t[t'_1, \ldots, t'_n]}$$

- *Abstraction*: Let x be not free in Γ.

$$\texttt{ABS} \quad \frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash (\lambda x.t_1) = (\lambda x.t_2)}$$

- *Type instantiation*: Let $t[\sigma_1, \ldots, \sigma_n / \alpha_1, \ldots, \alpha_n]$ be the result of substituting the types $\sigma_1, \ldots, \sigma_n$ for type variables $\alpha_1, \ldots, \alpha_n$ in t, and let none of the type variables α_1, ... , α_n occur in Γ, and no distinct variables in t become identified after the instantiation.

$$\texttt{TYPE\_INST}\ \frac{\Gamma_1 \vdash t_1 = t'_1 \cdots \Gamma_n \vdash t_n = t'_n \qquad \Gamma \vdash t[t_1, \ldots, t_n]}{\vdash \Gamma_1 \cup \cdots \cup \Gamma_n \cup \Gamma \vdash t[t'_1, \ldots, t'_n]}$$

- *Discharging an assumption*

$$\texttt{DISCH}\ \frac{\Gamma \vdash t_2}{\Gamma - \{t_1\} \vdash t_1 \implies t_2}$$

- *Modus ponens*

$$\texttt{MP}\ \frac{\Gamma_1 \vdash t_1 \implies t_2 \qquad \Gamma_2 \vdash t_1}{\Gamma_1 \cup \Gamma_2 \vdash t_2}$$

Axioms

The initial theory of HOL logic contains the following five axioms. All other theorems are based on these axioms through the inference rules presented above.

$$\texttt{BOOL\_CASES\_AX} \vdash_{\mathrm{def}} \forall t.\, (t = \mathsf{T}) \lor (t = \mathsf{F})$$
$$\texttt{IMP\_ANTISYM\_AX} \vdash_{\mathrm{def}} \forall t_1\, t_2.\, (t_1 \implies t_2) \implies (t_2 \implies t_1) \implies (t_1 = t_2)$$
$$\texttt{ETA\_AX} \vdash_{\mathrm{def}} \forall t.\, (\lambda x.\, t\, x) = t$$
$$\texttt{SELECT\_AX} \vdash_{\mathrm{def}} \forall P_{\alpha \to \mathbb{B}}.\, \forall x.\, P x \implies P(\varepsilon P)$$
$$\texttt{INFINITY\_AX} \vdash_{\mathrm{def}} \exists f_{I \to I}.\, \mathsf{OneOne}(f) \land \neg \mathsf{Onto}(f)$$

where OneOne and Onto are defined as follows:

$$\texttt{ONE\_ONE\_DEF} \vdash_{\mathrm{def}} \mathsf{OneOne}(f) = (\forall x_1\, x_2.(f(x_1) = f(x_2)) \implies (x_1 = x_2))$$
$$\texttt{ONTO\_DEF} \vdash_{\mathrm{def}} \mathsf{Onto}(f) = (\forall y.\exists x.\, y = f(x))$$

2.1.3 Using the Theorem Prover

The HOL system provides an environment for writing specifications and creating formal proofs of their properties. The work with a theorem prover is almost like programming. The role of modules is taken by theories in HOL. Each of the theories contains related types, constants, definitions and theorems, which can be reused. Like in programming, the design of a clean interface is the key to a good system structure.

The role of testing is taken by constructing proofs: When trying to show properties, all problems are discovered: definitions can be erroneous, proof tools may not not powerful enough, or the specification does not hold. In contrast to testing, a proof is always complete for the given property.

Additionally, there are libraries that contain custom proof tools. This part of the theorem prover is real programming: ML functions that manipulate terms, apply rules to theorems, or automatically construct proofs can be written to ease the proof development.

In this section, the user interface of the theorem prover is shown, i. e. how the user can extend existing theories and proof tools for his own needs to adapt the theorem prover to his application.

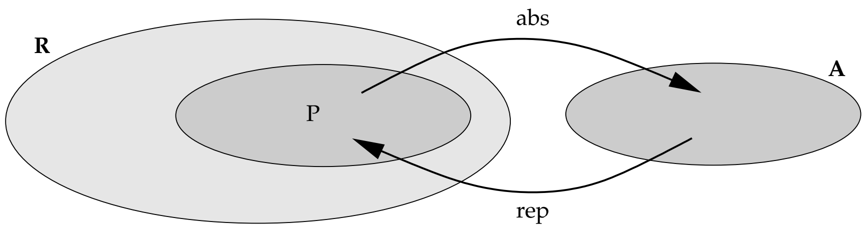

Fig. 5. Definition of New Types

Definition of Types

New types are always derived from existing ones (see Figure 5). Although the HOL library also provides other possibilities (inductive or algebraic definitions, quotients, records) to define new types, they are always reduced to the basic constructions presented in the following: From the set of existing types, a base type R is selected. With the help of a predicate P that is defined on elements of R, a subset is fixed that is the basis of the new type A. Each element inside P is mapped by the so-called abstraction function abs to an element of the new type. In the reverse direction, rep can be used to get the representative of a new element in the old type R.

New HOL types A are nonempty (as required by the base logic), i. e. there must always be an element for the new type. This must be explicitly proven for each new type when it is constructed, which is accomplished by proving $\exists r \in R.\ P(r)$.

For example[60], a type containing three elements $\mathbf{THREE}$ can be derived from the product of two Booleans $\mathbb{B} \times \mathbb{B}$. In order to get only three elements (and not the four possible ones of the base type), the predicate $P(x, y) = \neg(x \wedge y)$ is used to restrict the range. The result is a new type $\mathbf{THREE}$, which consists of the element set $\{abs(\mathsf{F}, \mathsf{F}), abs(\mathsf{F}, \mathsf{T}), abs(\mathsf{T}, \mathsf{F})\}$. For the non-emptiness proof, an arbitrary element of the three representatives is used as witness.

Definition of Terms

New functions and operators can be defined by the user as expected. The definition can either be an abbreviation or a (primitive or well-founded) recursive function. For recursive functions, termination must be proven if it is not obvious for the system[2]. Beside the condition that the right-hand side of the definition must not contain free variables, all occurrences of the defined term must have the same type (for recursive functions).
For example, the sign function for integers can be defined as follows:

$$\texttt{sgn}\ \vdash_{\text{def}} \operatorname{sgn}(x) = \textbf{if } x = 0 \textbf{ then } 0 \textbf{ else } (\textbf{if } x < 0 \textbf{ then } -1 \textbf{ else } 1)$$

An interesting question arises when functions are redefined. As HOL does not require that the left-hand side of a definition is unbound, a function can be given a new behaviour. Thus, theorems containing the old definitions generally do not hold for the new definition. However, they are not invalidated, but the identifier of the function is changed in all old theorems. Hence, several incarnations of a single function are possible, where only the last one is accessible by the specified name.

[2] As an example for a termination proof, see the Cohen-Sutherland line clipping algorithm presented in Section 4.5.2.

Theorems and Proofs

For a logician, a formal proof is a sequence, which contains as elements either axioms or propositions that follow from earlier members of the sequence by a rule of inference. The result of a proof is a theorem, which is given by the last element of the sequence.

In HOL, theorems are represented by instances of the ML data type `thm`, and rules are ML functions that take a number of these theorems as inputs and combine them to a new one. As there is no primitive constructor for the `thm` data type, rules are the only possibility to create new theorems. Thus, every value of type `thm` in the HOL system can be obtained by repeatedly applying primitive inference rules to axioms. [3]

The previous section listed the primitive inference rules, which are very low-level. To illustrate the rules, consider the following exemplary four rules, which are commonly used in proofs:

- *Undischarging* (`UNDISCH`): The left-hand side of an implication can be removed from the goal and added to the assumptions.
$$\frac{\Gamma \vdash t_1 \implies t_2}{\Gamma, t_1 \vdash t_2}$$
- *Symmetry of equality* (`SMY`): Both sides of an equation in a theorem can be exchanged.
$$\frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash t_2 = t_1}$$
- *Transitivity of equality* (`TRANS`): If a theorem states that if t_1 is equal to t_2, which is in turn equal to t_3, then t_1 is equal to t_3.
$$\frac{\Gamma_1 \vdash t_1 = t_2 \qquad \Gamma_2 \vdash t_2 = t_3}{\Gamma_1 \cup \Gamma_2 \vdash t_1 = t_3}$$
- *Rewriting* (`REWRITE_RULE`): One of the most important and most powerful inference rules is rewriting, which does a limited amount of automatic theorem-proving. It uses a list of equational theorems ($\Gamma \vdash t_1 = t_2$) to replace any subterms of an object theorem that match t_1 by the corresponding instance of t_2. Conditional and recursive rewriting is also supported.

Tactics

The style that is described in the last section is commonly referred to as a forward proof. For many applications, it is quite unnatural and too low-level to construct proofs in this way. Therefore, Milner invented the notion of tactics in the early 1970s, which represents an important advance in proof generating methodology.

Tactics are used in a goal-oriented proof. These proofs start with a goal $\Gamma \vdash^? t$, which consists of the same components as a theorem: Γ denotes the assumptions, whereas t represents the actual goal. A tactic is a program that takes such a goal $\Gamma \vdash^? t_1$ and splits it into a set of simpler subgoals $\Gamma_1 \vdash^? t_1, \ldots, \Gamma \vdash^? t_n$:

[3] In principle, there is another possibility, the function `mk_thm`, which can be used to arbitrarily create new theorems. However, these theorems are tagged, and these tags are propagated to all proofs that such theorems are used in. This mechanism allows the integration of external tools, like model checkers or sat solvers. Nevertheless, for each theorem, its origins are traceable, and the user knows on which tools he must trust for each theorem.

$$\begin{array}{c} \Gamma \vdash^? t \\ \hline\hline \Gamma_1 \vdash^? t_1 \quad \dots \quad \Gamma \vdash^? t_n \end{array}$$

Tactics keeps track why solving the subgoals will solve the complete goal. If all subgoals match already proven theorems, the proof can be constructed by them and the rule associated with the tactic.

So, a proof can be structured in a tree: In the root node, the original goal is specified. Tactics insert new nodes below the one containing the goal that the tactic is applied. When all leaves of the tree contain already proven theorems, these theorems are combined bottom-up in the tree with help of the rules associated with each tactic until the original goal is proven.

For instance, suppose the goal is to prove $A \wedge B$. Thus, it is sufficient to prove A and B, which is stated in the the $\wedge$-introduction, which corresponds to the HOL rule `CONJ`:

$$\frac{\Gamma_1 \vdash t_1 \qquad \Gamma_2 \vdash t_2}{\Gamma_1 \cup \Gamma_2 \vdash t_1 \wedge t_2}$$

Therefore, this justification can be used for the reduction of the goal $A \wedge B$ to the two subgoals A, as the HOL tactic `CONJ_TAC` implements:

$$\begin{array}{c} \Gamma \vdash^? t_1 \wedge t_2 \\ \hline\hline \Gamma_1 \vdash^? t_1 \quad \Gamma \vdash^? t_2 \end{array}$$

Libraries

Tactics are just one tool to construct proofs. Other important tools are simplification sets and conversions.

So-called simplification tactics define a general proof strategy based on simplification sets, which contain (amongst other components) rewriting theorems, associativity and commutativity theorems and decision procedures for a number of operations and definitions that are the scope of the simplification.

Conversions are ML functions that construct a theorem from a given term. For instance, the HOL library contains a conversion that takes a term describing an expression over the integers, reorders all parts of it to a normal form according to associativity and commutativity laws and finally proves the equivalence of the initial and the normal form. Conversions are very handy if some tactic should be only applied to a part of the goal. The conversion is given the desired part of the goal, for which it will generate an appropriate theorem. This can then be used for rewriting.

The HOL system features much more proof tools, which, however, will not discussed here. In the following, the formalisation of rational numbers is presented, which illustrates the descriptions of the HOL system given in this section.

2.2 Rational Numbers

The basis of all the verification work presented in this thesis are rational numbers. As the standard HOL library does not provide a theory for them, the first step with the theorem prover is their formalisation. Based on the theory of integers, fractions are introduced as pair of integers.

Subsequently, rational numbers are constructed as equivalence classes of them. The following paragraphs give an overview of the theory by showing its design goals and the most important theorems and tactics. In the meantime, the theory has been contributed to the standard HOL library, and is used in various other theories of the current version of the HOL system as e. g. in the embedding of the ACL2 theorem prover [57].

2.2.1 Definition of Fractions

The first step towards rational numbers is the definition of fractions. It is a matter of common knowledge that a fraction f consists of a pair of integers (a, b), whose components are known as the numerator (a) and the denominator (b) of f. The latter of the two components must not be zero; otherwise the fraction does not have a clear meaning. To make things easier, the sign of a fraction is always moved to the numerator. Thus:

$$f = \frac{a}{b} \text{ with } a, b \in \mathbb{Z},\ b > 0$$

The fractions theory introduces two constants 0_{frac} and 1_{frac}. Furthermore, it defines some basic operations on fractions: $\mathsf{fnmr}(f)$ returns the numerator, $\mathsf{fdnm}(f)$ the denominator and $\mathsf{fsgn}(f)$ the sign of a f, respectively. Two inverses are defined: the additive one $\mathsf{fainv}(f)$ and the multiplicative one $\mathsf{fminv}(f)$. Note that the domain of the multiplicative inverse is not restricted to non-zero fractions: $\mathsf{fminv}(0_{\mathsf{frac}}) = \frac{1}{0}$ is possible, albeit expressions containing this subexpression are almost unreducible. Addition $\mathsf{fadd}(f_1, f_2)$ and multiplication $\mathsf{fmul}(f_1, f_2)$ are defined intuitively. Subtraction $\mathsf{fsub}(f_1, f_2)$ and division $\mathsf{fdiv}(f_1, f_2)$ use these definitions and the additive and multiplicative inverse elements. In the following definitions, $\mathsf{fst}(p)$ denote the first component and $\mathsf{snd}(p)$ the second component of the underlying pair, respectively.

`frac_0` $\vdash_{\mathrm{def}} 0_{\mathsf{frac}} = \frac{0}{1}$
`frac_1` $\vdash_{\mathrm{def}} 1_{\mathsf{frac}} = \frac{1}{1}$
`frac_nmr` $\vdash_{\mathrm{def}} \mathsf{fnmr}(f) = \mathsf{fst}(\mathsf{rep}_{\mathsf{frac}}(f))$
`frac_dnm` $\vdash_{\mathrm{def}} \mathsf{fdnm}(f) = \mathsf{snd}(\mathsf{rep}_{\mathsf{frac}}(f))$
`frac_sgn` $\vdash_{\mathrm{def}} \mathsf{fsgn}(f) = \mathsf{sgn}(\mathsf{fnmr}(f))$
`frac_ainv` $\vdash_{\mathrm{def}} \mathsf{fainv}(f) = \frac{-\mathsf{fnmr}(f)}{\mathsf{fdnm}(f)}$
`frac_minv` $\vdash_{\mathrm{def}} \mathsf{fminv}(f) = \frac{\mathsf{fsgn}(f) \cdot \mathsf{fdnm}(f)}{|(\mathsf{fnmr}(f))|}$
`frac_add` $\vdash_{\mathrm{def}} \mathsf{fadd}(f_1, f_2) = \frac{\mathsf{fnmr}(f_1) \cdot \mathsf{fdnm}(f_2) + \mathsf{fnmr}(f_2) \cdot \mathsf{fdnm}(f_1)}{\mathsf{fdnm}(f_1) \cdot \mathsf{fdnm}(f_2)}$
`frac_mul` $\vdash_{\mathrm{def}} \mathsf{fmul}(f_1, f_2) = \frac{\mathsf{fnmr}(f_1) \cdot \mathsf{fnmr}(f_2)}{\mathsf{fdnm}(f_1) \cdot \mathsf{fdnm}(f_2)}$
`frac_sub` $\vdash_{\mathrm{def}} \mathsf{fsub}(f_1, f_2) = \mathsf{fadd}(f_1, \mathsf{fainv}(f_2))$
`frac_div` $\vdash_{\mathrm{def}} \mathsf{fdiv}(f_1, f_2) = \mathsf{fmul}(f_1, \mathsf{fminv}(f_2))$

In the pair library, there are theorems to access the components of a pair. Since fractions are derived from ordered pairs, similar theorems are provided by the fractions library: `NMR` and `DNM` reduce the terms $\mathsf{fnmr}(\frac{a}{b})$ and $\mathsf{fdnm}(\frac{a}{b})$. Both of them require that the denominator of the included fraction is positive. Otherwise, the simplification step cannot be performed. To show the required precondition, the conversions `FRAC_POS_CONV` and the tactic `FRAC_POS_TAC` can be used: they exploit that the denominators of fractions are positive (`FRAC_DNMPOS`) and the product of two positive numbers is positive (`INT_MUL_POS_SIGN`). Similarly, `FRAC_NOT0_CONV(t)` shows that a given term does not equal zero.

operator	precedence	description
$-$	*prefix*	additive inverse
$^{-1}$	*postfix*	multiplicative inverse
$+$	500	addition
$-$	500	subtraction
$\cdot$	600	multiplication
$/$	600	division
$<$	450	less than
$>$	450	greater than
$\leq$	450	less or equal
$\geq$	450	greater or equal

Fig. 6. Operators Defined over the Rationals

FRAC_NMR_CONV extracts the numerator of a fraction (with the help of NMR). If this conversion is given the term $\mathsf{fnmr}(\frac{a_1}{b_1})$, it will return the theorem $0 < b_1 \vdash \mathsf{fnmr}(\frac{a_1}{b_1}) = a_1$. FRAC_DNM_CONV works analogously. As it is very exhausting to do these simplification steps by hand, the fractions library provides FRAC_NMRDNM_TAC, which repeatedly applies the conversions to all appropriate terms of the goal.

Finally, FRAC_EQ_TAC is a tactic that is used in almost every proof involving fractions: To show that two fractions are equal, it is sufficient to show that both their numerators and denominators are equal.

2.2.2 Definition of Rational Numbers

Rational numbers are constructed as equivalence classes of fractions induced by the following relation $\simeq_{\text{rat}}$:

$$\texttt{rat\_equiv} \vdash_{\text{def}} f_1 \simeq_{\text{rat}} f_2 = (\mathsf{fnmr}(f_1) \cdot \mathsf{fdnm}(f_2) = \mathsf{fnmr}(f_2) \cdot \mathsf{fdnm}(f_1))$$

Two fractions f_1 and f_2 are equivalent under $\simeq_{\text{rat}}$, if and only if the fractions f_1' and f_2' that are obtained by the cancelling of f_1 and f_2 are equal. The following theorem asserts this intended meaning:

$$\begin{aligned}&\texttt{RAT\_EQUIV\_ALT} \vdash \simeq_{\text{rat}} a = \\ &\qquad \lambda x.(\exists bc.0 < b \wedge 0 < c \wedge (\mathsf{fmul}(a, \tfrac{b}{b}) = \mathsf{fmul}(x, \tfrac{c}{c})))\end{aligned}$$

After $\simeq_{\text{rat}}$ is proven to be an equivalence relation, the quotient type of rational numbers is defined with the help of the `quotient` theory [78]. Then, for each function defined over the fractions, a corresponding one for the rationals is defined. Figure 6 lists all functions with their corresponding operator symbols.

The following list gives the definition of all basic functions for the rationals. $\mathsf{mkRat}(f)$ denotes the abstraction of a fraction to a rational number, where $\mathsf{repRat}(r)$ is the inverse, i. e. it returns the representing fraction of a rational number.

rat_0 $\vdash_{\text{def}} 0 = \mathsf{mkRat}(0_{\mathsf{frac}})$
rat_1 $\vdash_{\text{def}} 0 = \mathsf{mkRat}(1_{\mathsf{frac}})$
rat_nmr $\vdash_{\text{def}} \mathsf{nmr}(r_1) = \mathsf{fnmr}(\mathsf{repRat}(r_1))$
rat_dnm $\vdash_{\text{def}} \mathsf{nmr}(r_1) = \mathsf{fdnm}(\mathsf{repRat}(r_1))$
rat_sgn $\vdash_{\text{def}} \mathsf{nmr}(r_1) = \mathsf{fsgn}(\mathsf{repRat}(r_1))$
rat_ainv $\vdash_{\text{def}} -r_1 = \mathsf{mkRat}(\mathsf{fainv}(\mathsf{repRat}(r_1)))$
rat_minv $\vdash_{\text{def}} r_1^{-1} = \mathsf{mkRat}(\mathsf{fminv}(\mathsf{repRat}(r_1)))$

rat_add $\vdash_{\text{def}} r_1 + r_2 = \mathsf{mkRat}(\mathsf{fadd}(\mathsf{repRat}(r_1), \mathsf{repRat}(r_2)))$
rat_sub $\vdash_{\text{def}} r_1 - r_2 = \mathsf{mkRat}(\mathsf{fsub}(\mathsf{repRat}(r_1), \mathsf{repRat}(r_2)))$
rat_mul $\vdash_{\text{def}} r_1 \cdot r_2 = \mathsf{mkRat}(\mathsf{fmul}(\mathsf{repRat}(r_1), \mathsf{repRat}(r_2)))$
rat_div $\vdash_{\text{def}} r_1 / r_2 = \mathsf{mkRat}(\mathsf{fdiv}(\mathsf{repRat}(r_1), \mathsf{repRat}(r_2)))$

The equivalence relation $\simeq_{\text{rat}}$ is proven to be a congruence relation for these operations. This is a very important precondition for the proof of all arithmetic rules. For instance, the theorems for the addition are as follows:

RAT_ADD_CONG1 $\vdash \mathsf{mkRat}(\mathsf{fadd}(\mathsf{repRat}(\mathsf{mkRat}(x)), y)) = \mathsf{mkRat}(\mathsf{fadd}(x, y))$
RAT_ADD_CONG2 $\vdash \mathsf{mkRat}(\mathsf{fadd}(x, \mathsf{repRat}(\mathsf{mkRat}(y)))) = \mathsf{mkRat}(\mathsf{fadd}(x, y))$

2.2.3 Arithmetic Rules for Rational Numbers

The rational numbers together with their operations form a field. In particular, they also form a ring. With these properties, various arithmetic rules are proven. Moreover, the conversions and tactics to simplify ring terms [60] of the `ringLib` can be used. They allow to transform ring terms in a normal form so that the equality of two expressions can be reduced to a syntactic equality test.

RAT_ADD_ASSOC $\vdash r_1 + (r_2 + r_3) = (r_1 + r_2) + r_3$
RAT_MUL_ASSOC $\vdash r_1 \cdot (r_2 + r_3) = (r_1 + r_2) \cdot r_3$
RAT_ADD_COMM $\vdash r_1 + r_2 = r_2 + r_1$
RAT_MUL_COMM $\vdash r_1 \cdot r_2 = r_2 \cdot r_1$
RAT_LDISTRIB $\vdash r_3 \cdot (r_1 + r_2) = r_3 \cdot r_1 + r_3 \cdot r_2$
RAT_RDISTRIB $\vdash (r_1 + r_2) \cdot r_3 = r_1 \cdot r_3 + r_2 \cdot r_3$
RAT_ADD_LID $\vdash 0 + r_1 = r_1$
RAT_ADD_RID $\vdash r_1 + 0 = r_1$
RAT_MULT_LID $\vdash 1 \cdot r_1 = r_1$
RAT_MULT_RID $\vdash r_1 \cdot 1 = r_1$
RAT_ADD_LINV $\vdash -r_1 + r_1 = 0$
RAT_ADD_RINV $\vdash r_1 + -r_1 = 0$
RAT_MUL_LINV $\vdash (r_1 \neq 0) \implies r_1^{-1} \cdot r_1 = 1$
RAT_MULRINV $\vdash (r_1 \neq 0) \implies r_1 \cdot r_1^{-1} = 1$
RAT_1_NOT_0 $\vdash 1 \neq 0$

The relations $\leq$ and $\geq$ are proven to be a total order, and $<$ and $>$ are shown to be the strict and total orders induced by them.

`RAT_LEQ_REF` $\vdash r_1 \leq r_1$
`RAT_LEQ_ANTISYM` $\vdash r_1 \leq r_2 \wedge r_2 \leq r_1 \implies (r_1 = r_2)$
`RAT_LEQ_TRANS` $\vdash r_1 \leq r_2 \wedge r_2 \leq r_3 \implies r_1 \leq r_3$
`RAT_LEQ_TOTAL` $\vdash r_1 \leq r_2 \vee r_2 \leq r_1$

`RAT_LES_REF` $\vdash \neg r_1 < r_1$
`RAT_LES_ANTISYM` $\vdash r_1 < r_2 \implies \neg r_2 < r_1$
`RAT_LES_TRANS` $\vdash r_1 < r_2 \wedge r_2 < r_3 \implies r_1 < r_3$
`RAT_LES_TOTAL` $\vdash r_1 < r_2 \vee (r_1 = r_2) \vee r_2 < r_1$

Various properties of the sign function sgn are proven, e. g. that the sign of a product is the product of the signs of the factors. With their help, the sign of an expression can be computed by the sign of the partial expressions.

`RAT_SGN_0` $\vdash \mathsf{sgn}(0) = 0$
`RAT_SGN_AINV` $\vdash -\mathsf{sgn}(-r_1) = \mathsf{sgn}(r_1)$
`RAT_SGN_MINV` $\vdash (r_1 \neq 0) \implies (\mathsf{sgn}(r_1^{-1}) = \mathsf{sgn}(r_1))$
`RAT_SGN_MUL` $\vdash \mathsf{sgn}(r_1 \cdot r_2) = \mathsf{sgn}(r_1) \cdot \mathsf{sgn}(r_2)$

The classical transformation of equations are proven to be correct. The first group of theorems describes modifications of both sides of an equation, while the second group deals with moving operations or terms from one side to another. For instance, the theorem `RAT_AINV_EQ` states that a minus sign in front of the left-hand side of the equation can be transferred to the right-hand side. With these theorems, equation transformations can be performed within the theorem prover.

`RAT_EQ_AINV` $\vdash (-r_1 = -r_2) = (r_1 = r_2)$
`RAT_EQ_RADD` $\vdash (r_1 + r_3 = r_2 + r_3) = (r_1 = r_2)$
`RAT_EQ_LADD` $\vdash (r_3 + r_1 = r_3 + r_2) = (r_1 = r_2)$
`RAT_EQ_RMUL` $\vdash (r_3 \neq 0) \implies ((r_1 \cdot r_3 = r_2 \cdot r_3) = (r_1 = r_2))$
`RAT_EQ_LMUL` $\vdash (r_3 \neq 0) \implies ((r_3 \cdot r_1 = r_3 \cdot r_2) = (r_1 = r_2))$

`RAT_AINV_EQ` $\vdash (-r_1 = r_2) = (r_1 = -r_2)$
`RAT_LSUB_EQ` $\vdash (r_1 - r_2 = r_3) = (r_1 = r_2 + r_3)$
`RAT_RSUB_EQ` $\vdash (r_1 = r_2 - r_3) = (r_1 + r_3 = r_2)$
`RAT_LDIV_EQ` $\vdash (r_2 \neq 0) \implies ((r_1/r_2 = r_3) = (r_1 = r_2 \cdot r_3))$
`RAT_RDIV_EQ` $\vdash (r_3 \neq 0) \implies ((r_1 = r_2/r_3) = (r_1 \cdot r_3 = r_2))$

Similar theorems are created for inequations. In contrast to the equations, there is the double number of theorems for the multiplication, since the sides of the equation must be swapped for negative factors.

RAT_LES_AINV $\vdash -r_1 < -r_2 = r_2 < r_1$
RAT_LES_LADD $\vdash (r_3 + r_1) < (r_3 + r_2) = r_1 + r_2$
RAT_LES_RADD $\vdash (r_1 + r_3) < (r_2 + r_3) = r_1 + r_2$
RAT_LES_LMUL_POS $\vdash 0 < r_3 \implies (r_3 \cdot r_1 < r_3 \cdot r_2) = r_1 < r_2)$
RAT_LES_LMUL_NEG $\vdash r_3 < 0 \implies (r_3 \cdot r_2 < r_3 \cdot r_1) = r_1 < r_2)$
RAT_LES_RMUL_POS $\vdash 0 < r_3 \implies (r_1 \cdot r_3 < r_2 \cdot r_3) = r_1 < r_2)$
RAT_LES_RMUL_NEG $\vdash r_3 < 0 \implies (r_2 \cdot r_3 < r_1 \cdot r_3) = r_1 < r_2)$

RAT_AINV_LES $\vdash -r_1 < r_2 = -r_2 < r_1$
RAT_LSUB_LES $\vdash (r_1 - r_2) < r_3 = r_1 < (r_2 + r_3)$
RAT_RSUB_LES $\vdash r_1 < (r_2 - r_3) = (r_1 + r_3) < r_2$
RAT_LDIV_LES_POS $\vdash 0 < r_2 \implies ((r_1/r_2 < r_3) = (r_1 < r_2 \cdot r_3))$
RAT_LDIV_LES_NEG $\vdash r_2 < 0 \implies ((r_1/r_2 < r_3) = (r_2 \cdot r_3 < r_1))$
RAT_RDIV_LES_POS $\vdash 0 < r_3 \implies ((r_1 < r_2/r_3) = (r_1 \cdot r_3 < r_2))$
RAT_RDIV_LES_NEG $\vdash r_3 < 0 \implies ((r_1 < r_2/r_3) = (r_2 < r_1 \cdot r_3))$

The rational numbers are proven to be dense, i. e. between two rational numbers that are not equal, a third number can always be found in between. For the proof, the average of the two given numbers is used as a witness.

RAT_DENSE_THM $\vdash r_1 < r_3 \implies \exists r_2.\, r_1 < r_2 \wedge r_2 < r_3$

2.2.4 Proof Tools

Moreover, conversions and tactics are provided that calculate normal forms of rational ring terms. With a simple syntactic test, the equality of two rational polynomials can be proven [64]. As an alternative, tactics are given that reduce expressions between rational numbers to expression between integers that can be handled by the integer decision procedures of the HOL system.

This heart of this reduction is the tactic RAT_CALCTERM_TAC(t): It transforms the expression t, which must be constructed by the rational arithmetic operators (see Figure 6) to a plain rational number of the form mkRat(f). Hence, a rational equation can be transformed to the normal form mkRat(f_1) = mkRat(f_2). The fractions f_1 and f_2 can now be tested for equivalence by comparing the mutual numerator-denominator product. The following rewrite rules are the core of this process:

`RAT_AINV_CALCULATE` $\vdash$
$$-\mathsf{mkRat}(f_1) = \mathsf{mkRat}(\mathsf{fainv}(f_1))$$
`RAT_MINV_CALCULATE` $\vdash$
$$\mathsf{fnmr}(f_2) \neq 0 \implies \mathsf{mkRat}(f_1)^{-1} = \mathsf{mkRat}(\mathsf{fminv}(f_1))$$
`RAT_ADD_CALCULATE` $\vdash$
$$\mathsf{mkRat}(f_1) + \mathsf{mkRat}(f_2) = \mathsf{mkRat}(\mathsf{fadd}(f_1, f_2))$$
`RAT_SUB_CALCULATE` $\vdash$
$$\mathsf{mkRat}(f_1) - \mathsf{mkRat}(f_2) = \mathsf{mkRat}(\mathsf{fsub}(f_1, f_2))$$
`RAT_MUL_CALCULATE` $\vdash$
$$\mathsf{mkRat}(f_1) \cdot \mathsf{mkRat}(f_2) = \mathsf{mkRat}(\mathsf{fmul}(f_1, f_2))$$
`RAT_DIV_CALCULATE` $\vdash$
$$\mathsf{fnmr}(f_2) \neq 0 \implies \mathsf{mkRat}(f_1)/\mathsf{mkRat}(f_2) = \mathsf{mkRat}(\mathsf{fdiv}(f_1, f_2))$$
`RAT_EQ_CALCULATE` $\vdash$
$$(\mathsf{mkRat}(f_1) = \mathsf{mkRat}(f_2)) = (\mathsf{fnmr}(f_1) \cdot \mathsf{fdnm}(f_2) = \mathsf{fnmr}(f_2) \cdot \mathsf{fdnm}(f_1))$$
`RAT_LES_CALCULATE` $\vdash$
$$(\mathsf{mkRat}(f_1) < \mathsf{mkRat}(f_2)) = (\mathsf{fnmr}(f_1) \cdot \mathsf{fdnm}(f_2) < \mathsf{fnmr}(f_2) \cdot \mathsf{fdnm}(f_1))$$

While `RAT_CALCTERM_TAC`(t) applies the calculation rules to the term t only, `RAT_CALC_TAC` applies it to all terms of type $\mathbb{Q}$ in the goal. If some of the rewrite rules above yield preconditions (i. e. positive denominators), the tactic automatically tries to prove them. In the case of failure, the proof obligations are added to the goal. Finally, `RAT_CALCEQ_TAC` performs all calculations and tries subsequently to prove all equalities by applying integer decision procedures.

The library `rat3Lib` defines two simplification sets: the first one `RAT3_ss` contains the usual arithmetic rules. The second simplification set `LOG3_ac_ss` also includes the associativity and commutativity theorems of the addition and multiplication. To ease transformations based on the associativity and commutativity of rational addition and multiplication, `RAT_AC_TAC`($\mathtt{e}_1 = \mathtt{e}_2$) takes an equation between two rational expressions e_1 and e_2. It tries to prove that both sides of the equation are equivalent using the associativity and commutativity laws and uses this obtained theorem to substitute e_1 by e_2 in the current goal.

This concludes the presentation of the rational numbers theory. The presented theorems and proof tools make it possible to reason about functions that use rational numbers as the geometry kernel, which is presented in the following. Further theories in this thesis will be given with less details. Similar to program code in C, they always contain a lot of technical details that are not needed for the understanding. Instead, the descriptions sum up the main results.

Chapter 3

Analytic Geometry

Classical geometry can be formalised in two ways: The first one is the axiomatic approach: All theorems about geometry are derived from Euclid's five axioms. The second approach is the construction of geometry with the help of a coordinate system. For the problem considered here, several points argue for the second way: the goal is the specification and verification of computational geometry algorithms, and these algorithms are generally based on a coordinate system. Moreover, the constructive way of introducing new types reflects the usual way followed in the HOL system. This may not have the beauty of the axiomatic way, but consistency is automatically guaranteed.

The second approach is often referred to as analytic geometry. In this domain, all geometric objects (like vectors, points, segments, rays, lines, curves, planes, circles, spheres etc.) have an algebraic counterpart: Points v are represented by n-tuples of numbers, and a set of points is described by a predicate P such that $P(v)$ holds whenever v belongs to the represented set. This is the viewpoint that computational geometry algorithms have. The work also follows this approach and builds a theory of geometry on top of basic algebraic structures.

The following section focuses underlying numbers, which are used for the components of vectors. Various issues with regard to the formalisation and implementation are discussed. Then, Section 3.3 presents the analytic geometry theory, which was created as a formal basis of the geometry kernel. Section 3.4 shows how this theory can be used for theorem proving, and Section 3.5 sketches the implementation of the geometry kernel.

3.1 Underlying Arithmetic

In mathematics, geometric problems are usually examined by the means of vectors, where points are given by tuples of real numbers. Nevertheless, this representation will not be used in the following for two reasons:

First, the gap between formalisation and implementation should be kept as small as possible. Although they can be formalised with a theorem prover [72], exact reals are far from any implementation. Thus, there would be a break between formalisation and implementation. Second, all problems in this thesis are related to polygons: Thus, all relevant objects can be represented by rational numbers — The only exception is the length of a vector. However, since the actual length of a vector is usually not of interest, but rather the comparison of distances, comparing the rational squares is sufficient. Other uses of real numbers are very rare in the domain of polygon-processing [98].

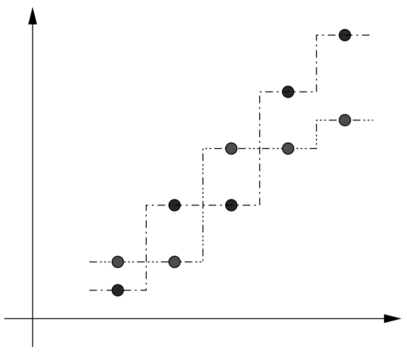

Fig. 7. Effects of Imprecise Floating Point Arithmetic

Rational numbers with arbitrary precision are used for the formalisation. They are implementable with the help of general purpose multiprecision libraries, like the GNU Multiprecision Library (GMP) [54]. All computations are then guaranteed to be correct, even though an order of magnitude slower than ordinary single-precision arithmetic, which is usable for prototypes. However, for real systems, this loss cannot be accepted. Thus, a reasonable efficiency requires approximations, which will be presented in Chapter 6.

The classical way to approximate reals or rationals in computers are floating-point numbers. Their fixed precision affects all computations and causes round-off errors that are due to overflows and numerical imprecisions. By restricting input data to single-precision numbers and using double-precision floating point numbers for intermediate results, the number of numerical errors can be reduced, but cannot eliminated in general. Thus, the use of the floating point kernel is unreliable and bears a certain amount of risk.

The basic problem is that the rounding of numbers follows fixed rules (round towards zero, round to nearest etc.) that are fatal for some algorithms. For example, due to rounding errors, the convex hull of a set of collinear points may consist of more than two points, or non-collinear lines may have more than one intersection point, which is illustrated by the following example [98]:

Consider the lines l_1 and l_2 given by the equations:

$$l_1 : y = \frac{9833}{9454} \cdot \quad \text{and} \quad l_2 : y = \frac{9366}{9005} \cdot x$$

Obviously, the slope of l_1 is larger than the slope of l_2, since at $x = 0$, $y_{l_1} = 0 = y_{l_2}$ and at $x = 9454 \cdot 9005$, $y_{l_1} = 9833 \cdot 9005 = 9366 \cdot 9454 + 1 = y_{l_2} + 1$. l_1 and l_2 intersect in the origin and l_1 is above l_2 for all positive values of x. However, calculating the coordinates for all multiples of 0.001 between 0 and 1 with the help of single precision floating point arithmetic leads to an astonishing result: The lines seem to be braided. Figure 7 illustrates this result.

As another example, consider the problem to find out whether the sequence of three points $\langle p_1, p_2, p_3\rangle$ is collinear or defines a left turn or a right turn. Rounding errors and overflows may lead the algorithms to believe that both sequences $\langle p_1, p_2, p_3\rangle$ and $\langle p_3, p_2, p_1\rangle$ define a left turn – a fact that is impossible.

Hence, special care must be taken when approximations are made in the implementation. In this chapter, arbitrary-precision numbers are used, and limited precision will be considered later. One reason for this is that a general approximation scheme cannot be fixed in this layer, since it depends on the geometric applications on top of it. Moreover, even if such a scheme existed, an inherent representation would complicate the formalisation by far and would make proof obligations very difficult. A distinction between the geometric part of an algorithm and the arithmetic part separates the concerns and limits the complexity of the verifications. The formalisation should be independent of the choice of arithmetic in the implementation.

So, all details that are related to arithmetic are factored out initially. They can be changed without affecting the correctness proof of the actual geometric algorithm (but not its implementation!). In the course of the verification of the algorithms, constraints can be determined for the involved computations to restrict the arbitrary arithmetic to a fixed-precision one.

3.2 Analytic Geometry Basics

This section introduces the basic analytic geometry [33] that will be used in the rest of the thesis. The terminology is fixed, and basic definitions and notions are given.

3.2.1 Basic Notions

The first thing that needs to be fixed in analytic geometry is the coordinate system. It maps a tuple of scalars to a point in an n-dimensional space. There are various systems, and each one is more or less appropriate for a particular problem. Two coordinate systems play an important role in computational geometry:

- *Cartesian Coordinates*: The Cartesian coordinate system is due to René Descartes, a French mathematician and philosopher, who published his ideas in 1637 in his book *La géométrie*. The modern Cartesian coordinate system in two dimensions is defined by two axes, at right angles to each other, forming a plane. The horizontal axis is labeled x, and the vertical axis is labeled y. After choosing a unit length and marking it off along each axis, a grid is obtained. With its help, a particular point p in the plane can now be specified by an ordered pair $(\mathsf{X}(p), \mathsf{Y}(p))$.
- *Homogeneous coordinates*: Homogeneous coordinates were originally introduced by August Ferdinand Möbius to make calculations possible in projective space. With their help, the basic transformations tranlation, scaling, rotation, shear and any combinations can be all expressed by a simple matrix multiplication. The homogeneous coordinates of a point p of projective space of dimension n are given by a $n+1$-tuple $(p_x, p_y, p_z, \ldots, p_w)$, where all components must not be zero at the same time. Homogeneous coordinates of a point are not unique: the same point can be specified by two triples that are multiples of each other.

In this thesis, the Cartesian Coordinates are used, because they are simpler in the formalisation of geometry.

Now that the coordinate system is fixed, the basic building block can be defined: A *vector* $v \in \mathbb{R}^2$ in the plane

$$v = \begin{pmatrix} \mathsf{X}(v) \\ \mathsf{Y}(v) \end{pmatrix}$$

consists of a tuple of scalars, which are usually called its *components*. $\mathsf{X}(v)$ is its x-coordinate and $\mathsf{Y}(v)$ its y-coordinate. The length and the norm of a vector $|v|$ are defined as follows:

$$\texttt{vec\_length} \vdash_{\text{def}} |v| = \sqrt{\mathsf{X}(v)^2 + \mathsf{Y}(v)^2}$$
$$\texttt{vec\_norm} \vdash_{\text{def}} ||v|| = \mathsf{X}(v)^2 + \mathsf{Y}(v)^2$$

The *orthogonal vector* $\mathsf{orth}(v)$ to another vector **v** is defined to be the vector of equal length that is turned clockwise by 90 degrees, which can be determined by

$$\texttt{vec\_orth} \vdash_{\text{def}} \mathsf{orth}(v) = \begin{pmatrix} \mathsf{Y}(v) \\ -\mathsf{X}(v) \end{pmatrix}$$

The dot product of two vectors v_1 and v_2 is the sum of the component-wise product:

$$\texttt{vec\_dprod} \vdash_{\text{def}} v_1 \circ v_2 = \mathsf{X}(v_1) \cdot \mathsf{X}(v_2) + \mathsf{Y}(v_1) \cdot \mathsf{Y}(v_2)$$

and has the following properties:

- $\cos(\alpha) = \frac{v_1 \circ v_2}{|v_1| \cdot |v_2|}$ (α is the angle between from v_1 to v_2)
- $v_1 \times v_2 = 0$ if and only if v_1 and v_2 are orthogonal

The *cross product* of two vectors $v_1, v_2 \in \mathbb{R}^3$ is defined as follows:

$$v_1 \times v_2 = \begin{pmatrix} \mathsf{Y}(v_1)\mathsf{Z}(v_2) - \mathsf{Z}(v_1)\mathsf{Y}(v_2) \\ \mathsf{Z}(v_1)\mathsf{X}(v_2) - \mathsf{X}(v_1)\mathsf{Z}(v_2) \\ \mathsf{X}(v_1)\mathsf{Y}(v_2) - \mathsf{Y}(v_1)\mathsf{X}(v_2) \end{pmatrix} \textbf{ with } v_1, v_2 \in \mathbb{R}^3$$

The length of the cross product $v_1 \times v_2$ can be interpreted as the unsigned area of the parallelogram having v_1 and v_2 as sides. A variant of it can be defined for the two-dimensional case. To this end, the z-component of each input vector is set to 0. The x and the y components of the cross product are 0, and the z-component is equal to the area of parallelogram denoted by v_1 and v_2. This operation can be abbreviated as follows:

$$\texttt{vec\_cprod} \vdash_{\text{def}} v_1 \times v_2 = \mathsf{X}(v_1) \cdot \mathsf{Y}(v_2) - \mathsf{Y}(v_1) \cdot \mathsf{X}(v_2)$$

A *point* p in the plane $\mathbb{R}^2$ is given by its position vector. Although there is a difference between vectors and points, they are not distinguished anymore, since they have the same base type (pairs of coordinates) and can be simply converted.

Lines are given by two different points p_1 and p_2 that the line crosses. Lines have an orientation, which will become interesting when talking about left or right of a line.

$$g : y(t) = p_1 + t \cdot v_g \textbf{ with } t \in \mathbb{R}$$

$v_g = (p_2 - p_1)$ is the *direction vector* of the line l. A segment $\overline{p_1 p_2}$ can be specified analogously by its two endpoints p_1 and p_2:

$$\overline{p_1 p_2} : y(t) = p_1 + t \cdot (p_2 - p_1) \textbf{ with } t \in [0, 1]$$

3.2.2 Intersection of Two Lines

Consider the computation of the intersection of two lines as an example. This problem is used throughout the chapter to illustrate the presented definitions and concepts. Moreover, this problem is a basic building block of many algorithms. It directly relies on the algebraic basis of computational geometry, and in the spirit of algebraic geometry, the geometric problem is transformed to an algebraic one in the following.

To compute the intersection point, it must be first checked whether such a point exists. Figure 8 shows the three cases: If the lines are parallel, they either do not share a common point (b) or

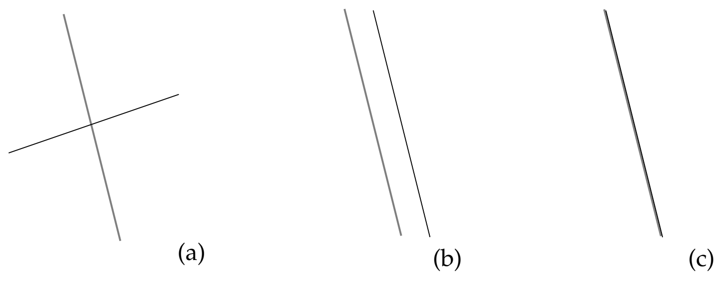

Fig. 8. Intersections of Two Lines

they are identical (c). Otherwise, there is a unique intersection point (a), which can be computed by solving the following equation system:

$$g_1 : y_1(t) = p_1 + t \cdot (p_2 - p_1) \textbf{ with } t \in \mathbb{R}$$
$$g_2 : y_2(t) = q_1 + t \cdot (q_2 - q_1) \textbf{ with } t \in \mathbb{R}$$

which can be transformed to

$$p_1 + \alpha_P(p_2 - p_1) = q_1 + \alpha_Q(q_2 - q_1)$$

α_P and α_Q can now be calculated as follows:

$$\begin{aligned}
& \alpha_P = \frac{(\mathsf{X}(q_1) - \mathsf{X}(p_1)) + \alpha_Q(\mathsf{X}(q_2) - \mathsf{X}(q_1))}{\mathsf{X}(p_2) - \mathsf{X}(p_1)} = \frac{(\mathsf{Y}(q_1) - \mathsf{Y}(p_1)) + \alpha_Q(\mathsf{Y}(q_2) - \mathsf{Y}(q_1))}{\mathsf{Y}(p_2) - \mathsf{Y}(p_1)} \\
\Leftrightarrow \quad & (\mathsf{X}(q_1) - \mathsf{X}(p_1))(\mathsf{Y}(p_2) - \mathsf{Y}(p_1)) + \alpha_Q(\mathsf{X}(q_2) - \mathsf{X}(q_1))(\mathsf{Y}(p_2) - \mathsf{Y}(p_1)) \\
& \quad = (\mathsf{Y}(q_1) - \mathsf{Y}(p_1))(\mathsf{X}(p_2) - \mathsf{X}(p_1)) + \alpha_Q(\mathsf{Y}(q_2) - \mathsf{Y}(q_1))(\mathsf{X}(p_2) - \mathsf{X}(p_1)) \\
\Leftrightarrow \quad & \alpha_Q\left[(\mathsf{X}(q_2) - \mathsf{X}(q_1))(\mathsf{Y}(p_2) - \mathsf{Y}(p_1)) - (\mathsf{Y}(q_2) - \mathsf{Y}(q_1))(\mathsf{X}(p_2) - \mathsf{X}(p_1))\right] \\
& \quad = (\mathsf{Y}(q_1) - \mathsf{Y}(p_1))(\mathsf{X}(p_2) - \mathsf{X}(p_1)) - (\mathsf{X}(q_1) - \mathsf{X}(p_1))(\mathsf{Y}(p_2) - \mathsf{Y}(p_1)) \\
\Leftrightarrow \quad & \alpha_Q = \frac{(p_2 - p_1) \times (q_1 - p_1)}{(q_2 - q_1) \times (p_2 - p_1)}
\end{aligned}$$

This leads to the following result for the intersection point S

$$S = (1 - \alpha_Q)q_1 + \alpha_Q q_2$$

Thus, the line intersection test returns two results: first, a Boolean value stating whether an intersection point exists and second, the intersection point itself (if it exists).

After a closer look to the equations, it becomes apparent that for the case that the two lines are collinear, the denominator of the fractions becomes zero and thus, they are undefined. Special cases like this one are typical for geometric computations and will be investigated more deeply in Chapter 4.

3.3 Formalisation of Analytic Geometry

3.3.1 Definitions

In the previous section, some basic notions of analytic geometry were introduced. In this section, it is shown how they can be formalised within the HOL system. Since the constructive approach of analytic geometry is used, the theory is based on the rational number theory of Section 2.2 and a vector is given by an ordered pair of rational numbers $\mathbb{Q} \times \mathbb{Q}$, encapsulated in a new HOL type $\mathbb{Q}^2$.

For this type, the following definitions are made: $\overrightarrow{0}$ denotes the zero vector, $\overrightarrow{1_\mathrm{x}}$ and $\overrightarrow{1_\mathrm{y}}$ denote the unit vectors of the x- and y-direction. The components of a vector $v = (\mathsf{X}(v); \mathsf{Y}(v))$ can be accessed by $\mathsf{X}(v)$ and $\mathsf{Y}(v)$, respectively. A vector can be mirrored, rotated and multiplied by a scalar. A pair of vectors can be added and subtracted:

$\texttt{vec\_mir} \vdash_{\mathrm{def}} -v = (-\mathsf{X}(v); -\mathsf{Y}(v))$
$\texttt{vec\_orth} \vdash_{\mathrm{def}} \mathsf{orth}(v) = (\mathsf{Y}(v); -\mathsf{X}(v))$
$\texttt{vec\_scale} \vdash_{\mathrm{def}} r \cdot v = (r \cdot \mathsf{X}(v); r \cdot \mathsf{Y}(v))$
$\texttt{vec\_add} \vdash_{\mathrm{def}} v_1 + v_2 = (\mathsf{X}(v_1) + \mathsf{X}(v_2); \mathsf{Y}(v_1) + \mathsf{Y}(v_2))$
$\texttt{vec\_sub} \vdash_{\mathrm{def}} v_1 - v_2 = (\mathsf{X}(v_1) - \mathsf{X}(v_2); \mathsf{Y}(v_1) - \mathsf{Y}(v_2))$

The dot product and the cross product of vectors are frequently used operations:

$\texttt{vec\_sprod} \vdash_{\mathrm{def}} v_1 \circ v_2 = \mathsf{X}(v_1) \cdot \mathsf{X}(v_2) + \mathsf{Y}(v_1) \cdot \mathsf{Y}(v_2)$
$\texttt{vec\_cprod} \vdash_{\mathrm{def}} v_1 \times v_2 = \mathsf{X}(v_1) \cdot \mathsf{Y}(v_2) - \mathsf{Y}(v_1) \cdot \mathsf{X}(v_2)$

The vectors $\mathbb{Q}^2$ form a vector space over the rational numbers $\mathbb{Q}$. As consequence of this, various arithmetic laws can be derived. For example, among these theorems are the following basic ones:

$\texttt{VEC\_ADD\_ASSOC} \vdash v_1 + (v_2 + v_3) = v_1 + v_2 + v_3$
$\texttt{VEC\_SCALE\_ASSOC} \vdash v_1 \cdot (v_2 \cdot v_3) = v_1 \cdot v_2 \cdot v_3$
$\texttt{VEC\_ADD\_COMM} \vdash v_1 + v_2 = v_2 + v_1$
$\texttt{VEC\_ADD\_LID} \vdash \overrightarrow{0} + v_1 = v_1$
$\texttt{VEC\_ADD\_RID} \vdash v_1 + \overrightarrow{0} = v_1$
$\texttt{VEC\_SCALE\_ID} \vdash 1 \cdot v_1 = v_1$
$\texttt{VEC\_ADD\_LINV} \vdash -v_1 + v_1 = \overrightarrow{0}$
$\texttt{VEC\_ADD\_RINV} \vdash v_1 + -v_1 = \overrightarrow{0}$
$\texttt{VEC\_ADD\_LDISTRIB} \vdash r_1 \cdot (v_1 + v_2) = r_1 \cdot v_1 + r_1 \cdot v_2$
$\texttt{VEC\_ADD\_RDISTRIB} \vdash (r_1 + r_2) \cdot v_1 = r_1 \cdot v_1 + r_2 \cdot v_1$

Vector operations can be reduced to operations on rational numbers by calculation theorems of the following form:

$\texttt{VEC\_ADD\_CALC} \vdash (r_1; r_2) + (r_3; r_4) = (r_1 + r_3; r_2 + r_4)$
$\texttt{VEC\_SUB\_CALC} \vdash (r_1; r_2) - (r_3; r_4) = (r_1 - r_3; r_2 - r_4)$
$\texttt{VEC\_SCALE\_CALC} \vdash r_1 \cdot (r_2; r_3) = (r_1 \cdot r_2; r_1 \cdot r_3)$
$\texttt{VEC\_SPROD\_CALC} \vdash (r_1; r_2) \circ (r_3; r_4) = r_1 \cdot r_3 + r_2 \cdot r_4$
$\texttt{VEC\_CPROD\_CALC} \vdash (r_1; r_2) \times (r_3; r_4) = r_1 \cdot r_4 - r_2 \cdot r_3$

For the cross product, the following rules (among others) can be used:

$\texttt{CPROD\_RDISTRIB} \vdash (v_1 + v_2) \times v_3 = (v_1 \times v_3) + (v_2 \times v_3)$
$\texttt{CPROD\_LSUM} \vdash (v_1 + v_2) \times v_2 = v_1 \times v_2$
$\texttt{CPROD\_RSUM} \vdash v_1 \times (v_1 + v_2) = v_1 \times v_2$
$\texttt{CPROD\_AINV} \vdash -(v_1 \times v_2) = v_2 \times v_1$

The cross product is a very important operation. For example, the linear dependency of two vectors is easily defined with it. For the linear dependency relation it is proven that it is an equivalence relation and that it commutes with various vector operations.

`vec_lindep` $\vdash_{\text{def}} \mathsf{lindep}(v_1, v_2) = (v_1 \times v_2 = 0)$
`VEC_LINDEP_REF` $\vdash \mathsf{lindep}(v_1, v_1)$
`VEC_LINDEP_SYM` $\vdash \mathsf{lindep}(v_1, v_2) = \mathsf{lindep}(v_2, v_1)$
`VEC_LINDEP_TRANS` $\vdash v_2 \neq \overrightarrow{0} \implies \mathsf{lindep}(v_1, v_2) \wedge \mathsf{lindep}(v_2, v_3) \implies \mathsf{lindep}(v_1, v_3)$

The area of a triangle and a quadrangle can also be defined with the cross product:

`area3` $\vdash_{\text{def}} \mathsf{triArea}(v_1, v_2, v_3) = \frac{1}{2} \cdot (v_2 - v_1) \times (v_3 - v_2)$
`area4` $\vdash_{\text{def}} \mathsf{quadArea}(v_1, v_2, v_3, v_4) = \mathsf{triArea}(v_1, v_2, v_3) + \mathsf{triArea}(v_1, v_3, v_4)$

Finally, the analytic geometry theory also includes important theorems of linear algebra like the two-dimensional case of Cramer's rule for the solution of a system of linear equations.

`CRAMERS_RULE` $\vdash \neg(v_1 \times v_2 = 0) \implies$
$$((v_0 = r_1 \cdot v_1 + r_2 \cdot v_2) =$$
$$(r_1 = (v_0 \times v_2)/(v_1 \times v_2)) \wedge (r_2 = (v_1 \times v_0)/(v_1 \times v_2)))$$

Similarly, to the rational numbers, various tactics and conversions are provided for simplifying and deciding vector expressions. For instance, `VEC_CALCEQ_TAC` transforms equations on rational vectors to two equations on vectors (or simplifies rational equations involving vector operations) and subsequently invokes the rational numbers decision tactic `RAT_CALCEQ_TAC`.

3.4 Algebraic Geometry Theorem Proving

Formal reasoning about geometric problems has a long tradition in automated theorem proving. Hilbert already gave a general proof strategy for affine geometry in his classic book [74]. A decision procedure for analytic geometry using quantifier elimination was presented in 1951 by Tarski [128]. However, it was merely a theoretical result without any practical relevance. The improvements made by Collins [32] moved the decision procedure a bit more into practice, but did not change the situation significantly.

The fact that geometry is one of the most successful areas of automated theorem proving is due to Wu [25, 132, 133, 134]. His algebraic method became very popular and allowed the proof of hundreds of geometry theorems [26]. It is based on the general reduction of geometric proofs to algebraic ones on base of analytic geometry, which is presented in the following.

For the verification of polygon processing algorithms, geometry theorem proving can be used as a basic building block. Algorithms are often based on certain geometric properties that must be checked for a correct execution of the overall algorithm. However, it is important to distinguish between geometry theorem proving and reasoning about geometric algorithms, which involves in addition to the geometry dynamic parts, e. g. control and data structures, of the algorithms.

3.4.1 Reduction to Algebraic Problems

For instance, consider a parallelogram given by the points p_1, p_2, p_3 and p_4 as shown in Figure 9. The diagonals intersect in a point p_5. Apparently, p_5 is in the middle of each diagonal, which should be the proof obligation in the following.

In the classical analytic proof, the geometric situation is first expressed in an algebraic way: Assume without loss of generality that $p_1 = (0; 0)$, $p_2 = (u_1; 0)$ and $p_3 = (u_2; u_3)$ are fixed. The

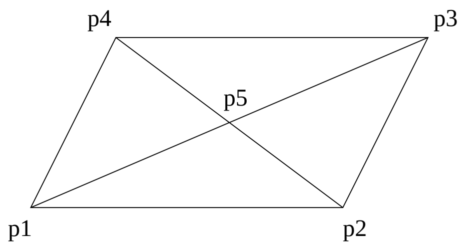

Fig. 9. Parallelogram and its Diagonals

fourth point $p_4 = (x_2; x_1)$ is dependent, since the resulting object forms a parallelogram[1]. This dependency can be formalised by the following two equations that become assumptions of the proof goal:

$$h_1 : u_1 x_1 - u_1 u_3 = 0$$
$$h_2 : u_3 x_2 - (u_2 - u_1) x_1 = 0$$

The equation h_1 states that $\overline{p_1 p_2}$ is parallel to $\overline{p_4 p_3}$, and h_2 states that $\overline{p_4 p_1}$ is parallel to $\overline{p_3 p_2}$. After x_1 and x_2 are defined, the positions of the point $p_5 = (x_4; x_3)$ is fixed analogously. Hence, two further assumptions are obtained:

$$h_3 : x_1 x_4 - (x_2 - u_1) - u_1 x_1 = 0$$
$$h_4 : u_3 x_4 - u_2 x_3 = 0$$

The proof goal is also encoded in an algebraic form. The length of both halfs of the diagonal from p_1 to p_3 is equal if and only if the following equation g holds. This is expressed as follows:

$$g : 2u_2 x_4 + 2u_3 x_3 - u_3^2 - u_2^2 = 0$$

With the help of basic algebraic methods or Cramer's rule (see 3.3), the goal can be easily proven. From the first two equations, $x_1 = u_3$ and $x_2 = u_2 - u_1$ are derived, the last two ones give $x_3 = \frac{1}{2} u_3$ and $x_4 = \frac{1}{2} u_2$. These parameters can be substituted in g, which results to $g = 0$, the proof goal.

The theorem, however, is not generally valid. In the course of the proof, when applying Cramer's rule or transforming the equations, several side conditions are assumed. The problem has already appeared in Section 2.2: Some transformations, e. g. the multiplication of both sides of an equation, requires that the factor used is not zero. Analogously, Cramer's rule can be only applied if both equations are not linearly dependent (leading to a zero in the denominator of the rule). For the parallelogram, the solutions of x_1 and x_4 require that neither u_1 nor u_3 are zero. For these situations, the parallelogram would be flat, without a meaningful definition of the diagonals.

When using the arithmetic rules presented in the previous chapter, these degenerate conditions become automatically apparent and show under which conditions the conclusion of the theorem is really valid.

3.4.2 Wu's Approach

Wu realised the importance of the nondegenerate conditions [132] and developed an approach that automatically deals with them. He generalised the approach presented in the previous section and developed a generally applicable scheme for all properties based on equations.

[1] To distinguish free and dependent parameters, their names are given either given by u_i or x_i.

Assumptions h_1 to h_n and the proof goal g are given by polynomial equations over free parameters u_1 to u_d and dependent parameters x_1 to x_t:

$$\begin{aligned} h_1(u_1,\ldots,u_d,x_1,\ldots,x_t) &= 0 \\ h_2(u_1,\ldots,u_d,x_1,\ldots,x_t) &= 0 \\ &\vdots \\ h_n(u_1,\ldots,u_d,x_1,\ldots,x_t) &= 0 \\ g(u_1,\ldots,u_d,x_1,\ldots,x_t) &= 0 \end{aligned}$$

As a first step, these equations are triangulated so that each one introduces only one new dependent variable, i. e. each assumption h_i only depends on the variables x_1 to x_i. By consecutive pseudo divisions the validity of the goal is checked and simultaneously, nondegenerate conditions are automatically derived.

The proofs produced by Wu's approach are very low-level and are thus not traceable. In an interactive theorem prover they are only suited for a decision procedure. However, for the polygon processing verification, Wu's method was not used, since its implementation inside the theorem prover would involve a lot of underlying mathematics. Complicated polynomial operations like the pseudo divisions and remainders must be formalised and properties for them have to be proven. This effort is not justified for the polygon-processing library, as many goals can be also solved manually without much effort. This is even more the case if proof tools for constructive proofs (see Section 4.6) are available. These proof tools are more suited in the context of interactive theorem proving, as they generate proofs at a higher level of abstraction.

3.5 Implementation of a Rational Geometry Kernel for a Software Library

All the definitions of this chapter are implemented in a geometry kernel, which is used as the core of the polygon-processing library. In the following code fragments, all types and functions of the library can be recognised by the prefix `epp`.

The basic building block is again a vector. It is implemented as a structure containing two rationals of type `epp_rat`. This type encapsulates the rational numbers, which are implemented preliminarily (before rounding is discussed later) with arbitrary precision rationals as provided by the GNU Multiprecision Library (GMP) [54].

```
struct epp_vec_t { epp_rat x; epp_rat y; };
typedef struct epp_vec_t epp_vec;
```

The library contains all vector function that are formalised above, among them for example the following ones:

```
void epp_vec_norm( epp_rat *r0, const epp_vec *v1 );
void epp_vec_add( epp_vec *v0, const epp_vec *v1, const epp_vec *v2 );
void epp_vec_sub( epp_vec *v0, const epp_vec *v1, const epp_vec *v2 );
void epp_vec_scale( epp_vec *v0, const epp_vec *v1, const epp_rat *r1 );
void epp_vec_orth_ccw( epp_vec *v0, const epp_vec *v1 );
void epp_vec_orth_cw( epp_vec *v0, const epp_vec *v1 );
void epp_vec_sprod( epp_rat *r0, const epp_vec *v1, const epp_vec *v2 );
void epp_vec_cprod( epp_rat *r0, const epp_vec *v1, const epp_vec *v2 );
void epp_vec_solve_eqns( epp_rat *r0, epp_rat *r1,
   const epp_vec *v0, const epp_vec *v1, const epp_vec *v2 );
```

All these functions are implemented straightforwardly. The results of an operation are not returned by the function, but are written to the first argument. This has technical reasons: To avoid memory leaks, the memory to store the result is allocated by the calling function. Although the polygon-processing library runs with static memory, the prototype implementation with the GMP data structures needs dynamic memory.

Similarly to the interface, the implementation of the operations follows the analytic geometry formalisation of Section 3.3.

Chapter 4

Geometric Primitives

The line intersection computation, which was presented in Section 4.1.2, is a typical example of a basic operation that is used in many geometric problems. Many computational algorithms rely on a small number of so-called primitives, i. e. basic operations of very limited functionality that can be commonly reused. Like the line intersection problem, they generally use the foundations of computational geometry and solve the geometric problem by transforming it into an algebraic one. For this reason, these primitives form an abstraction layer on top of the algebraic foundations. Hiding the details for higher-level algorithms is not only desired for an implementation: It also becomes very important when reasoning about the correctness of algorithms, since low-level proofs are both unreadable and unmaintainable for non-trivial algorithms.

This chapter first introduces in Section 4.1.1 geometric primitives as they are usually given in the literature. Section 4.2 describes the problem of degenerate cases and previously proposed solutions. Section 4.3 introduces three-valued logic, and Section 4.4 shows how it can be applied to the problem. The next sections focus on the verification. Symmetries and how they can be exploited in proofs are the topic of Section 4.5. Section 4.6 presents how the constructions of dependant objects can be exploited. Finally, Section 4.7 sketches the implementation of the primitives in the developed polygon-processing library.

4.1 A First Approach

4.1.1 Two-Valued Primitives

This section defines some primitives that deal with linear geometric objects. A special class of the primitives, which will be called *topological primitives* in the following, tests some input objects and classifies them according to a predefined number of cases. For example, the simplest class compares the position of two points:

Definition 4 (Relative Position of Two Points). *A point p is* left from *another point q if and only if the x-coordinate of p is less than the x-coordinate of q, formally $\chi_{\mathsf{Left}}(p,q) := \mathsf{X}(q) - \mathsf{X}(p) < 0$. Analogously, a point p is* below *q if and only if the corresponding condition holds for the y-coordinates: $\chi_{\mathsf{Below}}(p,q) := \mathsf{Y}(q) - \mathsf{Y}(p) < 0$.*

Another very important primitive in computational geometry algorithms is the orientation of three points, i. e. whether a sequence of three points forms a left or a right turn.

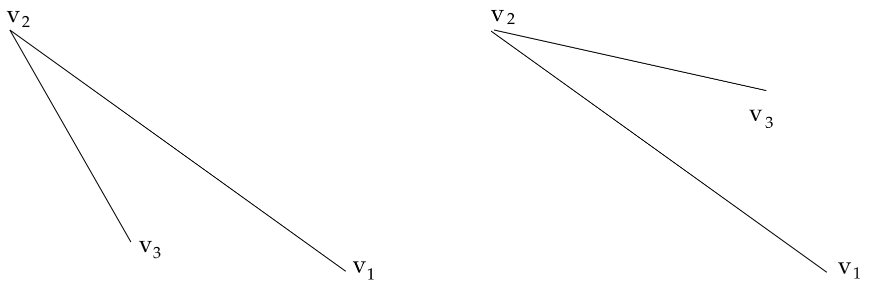

Fig. 10. Orientation of Three Points

Definition 5 (Orientation of Three Points). *The points p, q and r form a* left turn *(left-hand side of Figure 10) if and only if the following determinant is positive:*

$$\chi_{\mathsf{lturn}}(p,q,r) := \begin{vmatrix} \mathsf{X}(p) & \mathsf{Y}(p) & 1 \\ \mathsf{X}(q) & \mathsf{Y}(q) & 1 \\ \mathsf{X}(r) & \mathsf{Y}(r) & 1 \end{vmatrix}$$

Three points p, q and r are said to be collinear *if and only if $\chi_{\mathsf{lturn}}(p,q,r) = 0$.*

In the previous chapter, linear dependency was already defined. However, for the sake of completeness, the definition is restricted here:

Definition 6 (Linear dependency). *Two vectors v_1 and v_2 are linearly dependent if their cross product is zero:*

$$\chi_{\mathsf{lindep}}(v_1, v_2) = (v_1 \times v_2 = 0)$$

Two lines are parallel *if their direction vectors are linearly dependent.*

4.1.2 Intersection of Two Line Segments

Line segment intersection is a very important primitive for many computational geometry algorithms. In the following, the approach is illustrated by the line intersection example: For two given lines l_1 and l_2, it should be checked whether they intersect or not, and if so, the intersection point should be calculated.

Specification of Line Intersection

The intersection point of two line segments l_1 and l_2 is characterised in that it belongs to both lines:

$$\texttt{is\_line\_int} \vdash_{\text{def}} \mathsf{isLineIntersect}(l_1, l_2, v_1) = \mathsf{onLine}(l_1, v_1) \wedge \mathsf{onLine}(l_2, v_1)$$

However, such an intersection point does not always exist. Thus, the following predicate only states whether such a point exists or not.

$$\texttt{exists\_line\_int} \vdash_{\text{def}} \mathsf{existsLineIntersect}(l_1, l_2) = \exists v_1.\mathsf{isLineIntersect}(l_1, l_2, v_1)$$

Implementation of Line Intersection

The following functions give an implementation of the check and the calculation: Two lines have a single intersection point if and only if they are linearly independent, which is formalised in the function doLineIntersect :

$$\texttt{line\_parallel} \vdash_{\text{def}} \mathsf{parallel}(l_1, l_2) = \mathsf{lindep}(\mathsf{dirvec}(l_1), \mathsf{dirvec}(l_2))$$
$$\texttt{do\_line\_int} \vdash_{\text{def}} \mathsf{doLineIntersect}(l_1, l_2) = \neg\mathsf{parallel}(l_1, l_2)$$

If an intersection point exists, it is calculated by the function $\mathsf{lineIntersect}(\cdot,\cdot)$ for both lines and segments.

$$\texttt{vec\_on\_line} \vdash_{\text{def}} \mathsf{vecAt}\,(l_1, r_1) = \mathsf{beg}(l_1) + r_1 \cdot \mathsf{dirvec}(l_1)$$
$$\texttt{line\_intersect} \vdash_{\text{def}}$$
$$\mathsf{lineIntersect}(l_1, l_2) = \mathsf{vecAt}\left(l_2, \frac{(\mathsf{beg}(l_1)) - \mathsf{beg}(l_2) \times \mathsf{dirvec}(l_1))}{\mathsf{dirvec}(l_2) \times \mathsf{dirvec}(l_1)}\right)$$

Verification of Line Intersection

To verify the implementation, the following theorems must be checked:

$$\texttt{LINEINT\_EXIST\_CORRECT} \vdash \mathsf{parallel}(l_1, l_2) \implies$$
$$\mathsf{existsLineIntersect}(l_1, l_2) = \mathsf{doLineIntersect}(l_1, l_2)$$
$$\texttt{LINEINT\_CALC\_CORRECT} \vdash \mathsf{existsLineIntersect}(l_1, l_2) \implies$$
$$(\forall v.(\mathsf{isLineIntersect}(l_1, l_2, v) = \mathsf{T}) = (v = \mathsf{lineIntersect}(l_1, l_2)))$$

They state that if two lines are not parallel, v is the intersection of two lines if and only if v is the result of the line intersection function. The proofs for these theorems are done by rewriting with the definitions and finally applying theorem CRAMERS_RULE.

By an analogous proof with swapped line arguments, the symmetry of the line intersection procedure can be additionally derived. The assumption is necessary. Otherwise, the intersection point would not be defined, since its computation involves a division by zero.

$$\texttt{LINE\_INTERSECT\_SYM} \vdash$$
$$\neg\mathsf{parallel}(l_1, l_2) \implies (\mathsf{lineIntersect}(l_1, l_2) = \mathsf{lineIntersect}(l_2, l_1))$$

Now, consider the problem of computing the intersection of two line *segments*. It is clear that the two segments only have an intersection point if the corresponding two lines have an intersection point. Additionally, it must be checked whether the endpoints of both segments are on different sides of the other segment. This can be done with the help of the primitives defined above.

In general position, two segments do either not intersect or they intersect at a point interior to both segments. Additional special cases must be considered: Two intersecting segments may also touch, overlap, share a common endpoint or have one segment endpoint interior to the other segment – and each case exists in different variations. Figure 11 lists some of the possible cases. In addition to the complexity of the high number of different cases, it is not clear what a good definition should be for some cases: Consider cases (c) and (d): Should two touching segments count as an intersection or not? This clearly depends on the algorithm that uses this primitive: Some algorithms assume segments to be open, others use closed segments. This example illustrates that establishing consistent definitions is difficult even for simple operations.

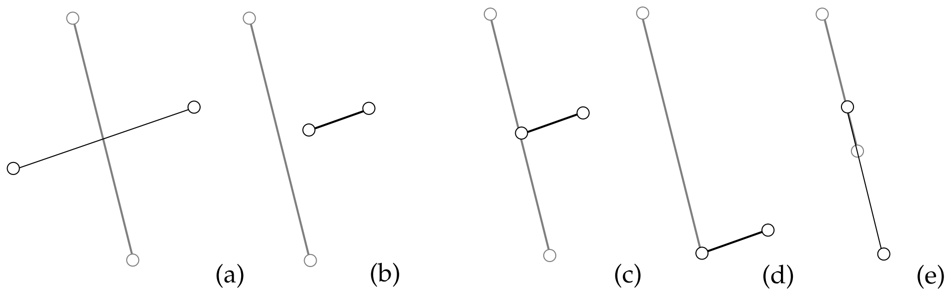

Fig. 11. Cases for the Intersection of Two Line Segments

Hence, two definitions of each primitive presented in Section 4.1.1 are needed. Alternatively, since the primitives are somehow antisymmetric, they can be used with negation or different parameter orderings; e. g. $\mathsf{lturn}(v_1, v_2, v_3)$ states that the sequence $\langle v_1; v_2; v_3\rangle$ forms a left turn, whereas $\neg\mathsf{lturn}(v_3, v_2, v_1)$ says that the same sequence $\langle v_1; v_2; v_3\rangle$ is collinear or forms a right turn.

4.2 Degenerate Cases

4.2.1 Challenges by Degenerate Cases

The line segment intersection problem is a good example for degenerate cases. Algorithms that can be found in textbooks about computational geometry usually neglect special cases: Depending on the algorithm, several preconditions are assumed, e. g. that no points coincide, that given lines are not parallel or that no three lines intersect in a common point. Especially in research papers, authors abstract from some or all of these so-called *degenerate cases*, which pose a well-known problem to computational geometry algorithms. There are several reasons for this: Either, the researchers want to focus on the core of their algorithms not bothering the reader with (in their eyes) tedious details or they are not aware of all the special cases that may appear. From the theoretical point of view, this approach is generally justified, as the degenerate cases do neither change the general structure of a geometric algorithm nor its asymptotic complexity.

Regardless which reason leads to the sketchy definitions of the algorithms, the degenerate cases cannot be neglected when implementing the algorithms. A precise handling of all the tedious details is a fundamental point in real world applications. This is even more the case for safety-critical embedded systems, which are the application domain of this thesis.

Degenerate cases have been a subject for many researchers in the area of computational geometry. Since they often require a substantial amount of additional effort, it is not be desirable to explicitly refer to all degenerate cases of complex algorithms. The main problem is that non-trivial algorithms may have high numbers of degenerate cases due to the combination of several basic ones. It is clear that this number correlates with the length of the implementation and the amount of time to maintain it.

4.2.2 Symbolic Perturbations

Considering this problem, some researchers try to completely eliminate the problem of degenerate cases by perturbing inputs, so that all predicates are guaranteed to be evaluated for general cases

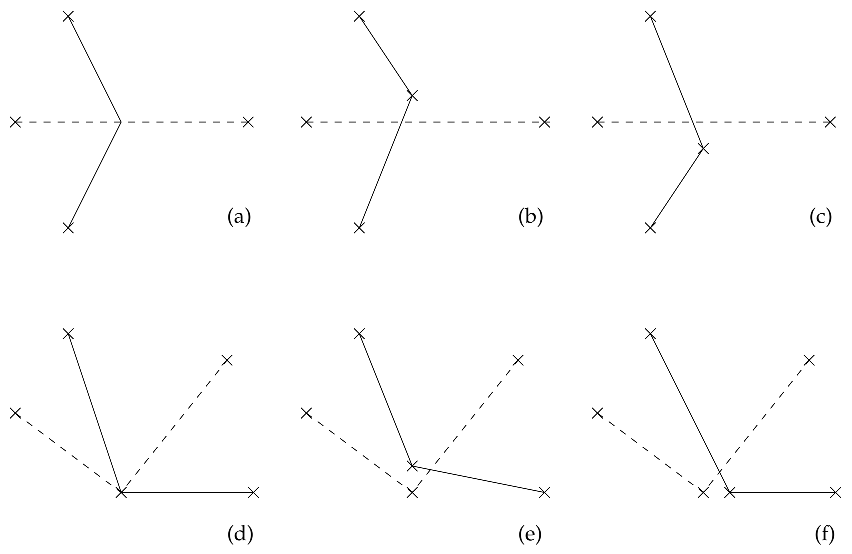

Fig. 12. Resolving Degenerate Cases

only. For example, situation (a) in Figure 12 is either mapped to (b) or (c). Analogously, the common point of both sequences in (d) is moved to resolve the situation like in (e) or (f).

This approach is motivated by the fact that the set of degenerated cases has always a smaller dimension than the actual problem. There are always infinitly many points in the ϵ-sphere of a degenerate point p that have nondegenerate positions. Otherwise, these cases would not be degenerate. A slight change of the inputs allows the algorithms to neglect degenerated cases. The resulting programs are then smaller and more robust: The tedious treatment of many special cases can be replaced by a single perturbation step before the computation.

However, this approach is not a real solution: As the algorithm is not really able to handle degenerate cases, instead it solves a similar problem. This may be acceptable, if the new problem is irrelevant w. r. t. a given application. For example, in the domain of computer graphics, modifications that lead to an error less than a pixel can be seen to be irrelevant. Nevertheless, such an upper bound for a tolerable error cannot be defined in general - and in particular for safety-critical embedded systems.

A solution to this problem are *symbolic perturbations* [39, 43, 45, 135, 136, 137, 138]: Such perturbations are only conceptual and do not really modify the inputs. They are given in terms of an infinitesimally small ϵ, which does not need to be fixed exactly. Only the following two conditions must be met [39]:

- The perturbation must lead to nondegenerate results of the primitives.
- The perturbation must be small enough to retain all nondegenerate properties of the original inputs.

Formally, these conditions require the perturbation to be strictly monotonic if the degenerate case is seen to be smaller than all other cases. Clearly, the result of an operation on perturbed inputs should be the same as on original data, since the perturbation is intended to be only symbolical.

Edelsbrunner and Mücke

In [39] Edelsbrunner and Mücke defined a perturbation scheme that fulfils the conditions above. Moreover, they designed it with runtime performance in mind: Its computational overhead is kept as low as possible (or negligible as they say).

They handle degenerate cases transparently by modifying the basic geometric primitives to operate on perturbed input data instead of the original one. With the help of this modification, ties are consistently broken. Intuitively, each point is replaced with a symbolically perturbed point, given by a vector of polynomials of an infinitesimal small number ϵ. For example, consider n geometric points with d parameters each:

$$\{p_0, p_1, \ldots, p_{n-1}\} \textbf{ with } p_i = (\pi_{i,1}, \pi_{i,2}, \ldots, \pi_{i,d}) \textbf{ for } 0 \leq i \leq n-1$$

Thus, each point is attributed a unique index between 0 und $n-1$. The perturbation for a point p_i is given by the following tuples of polynomials:

$$p_i(\epsilon) = (\pi_{i,1}(\epsilon), \pi_{i,2}(\epsilon), \ldots, \pi_{i,d}(\epsilon)) \textbf{ for } \pi_{i,j}(\epsilon) = \pi_{i,j} + \epsilon(i,j) \textbf{ with } 0 \leq i \leq n-1, 1 \leq j \leq d$$

$\epsilon(i,j)$ must be a polynomial, which goes to 0 if ϵ also goes to zero ($\lim_{\epsilon \to 0} = 0$). Edelsbrunner and Mücke choose the following concrete value:

$$\epsilon(i,j) = \epsilon^{2^{i \cdot \delta - j}} \textbf{ for } 0 \leq i \leq n-1, 1 \leq j \leq d, \delta \geq d$$

All primitives of Section 4.1.1 can be formulated as the computation of a matrix sign. Substitution of the symbolically perturbed points in a topological primitive results in a polynomial in the variable ϵ with coefficients determined by the original points. As a consequence of the elegant choice of the perturbation, the sign of the expression is given by the sign of the first nonzero coefficient, where coefficients are taken in increasing order of powers of ϵ.

The perturbation is illustrated with the help of the following matrix:

$$\Delta_2 = \begin{pmatrix} \pi_{i,1} & \pi_{i,2} \\ \pi_{j,1} & \pi_{j,2} \end{pmatrix}$$

The perturbation extends the components of the matrix to

$$\Delta_2(\epsilon) = \begin{pmatrix} \pi_{i,1} + \epsilon(i,1) & \pi_{i,2} + \epsilon(i,2) \\ \pi_{j,1} + \epsilon(j,1) & \pi_{j,2} + \epsilon(j,2) \end{pmatrix}$$

To abbreviate, define $\epsilon(\cdot) = 1$ and $\epsilon((i_1, j_1), \ldots, (i_k, j_k)) = \prod_{\nu=1}^{k} \epsilon(i_\nu, j_\nu)$ to be the k-ary ϵ product. Moreover, let $i < j$. The determinant of the matrix can be computed as follows:

$$\begin{aligned}
\det \Delta_2(\epsilon) = &+ \det \begin{pmatrix} \pi_{i,1} & \pi_{i,2} \\ \pi_{j,1} & \pi_{j,2} \end{pmatrix} \cdot \epsilon() && (1)\\
&- \pi_{j,1} \cdot \epsilon(i,2) && (2)\\
&+ \pi_{j,2} \cdot \epsilon(i,1) && (3)\\
&+ \pi_{i,1} \cdot \epsilon(j,2) && (4)\\
&+ 1 \cdot \epsilon((j,2),(i,1)) && (5)\\
&- \pi_{i,2} \cdot \epsilon(j,1) && (6)\\
&- 1 \cdot \epsilon((j,1),(i,2)) && (7)
\end{aligned}$$

The terms are ordered in decreasing order of ϵ. In the course of the computation, the first term that is not zero determines the result of the primitive, since the absolute value of the sum of all following terms are guaranteed to be smaller. The fifth part guarantees that such a term exists: It is always 1, regardless of the inputs. Therefore, all following parts do not have an influence on the sign of $\det \Delta_s(\epsilon)$. Moreover, the first part of the determinant is equal to the determinant of the unperturbed matrix. This ensures that the perturbation does not change the result of the general situation.

t	size	$\det M_t^{\Delta_2}$	ϵ_t
0	$2 \cdot 2$	$\det \begin{pmatrix} \pi_{i,1} & \pi_{i,2} \\ \pi_{j,1} & \pi_{j,2} \end{pmatrix}$	$\epsilon(\cdot)$
1	$1 \cdot 1$	$-\pi_{j,1}$	$\epsilon(i,2)$
2	$1 \cdot 1$	$+\pi_{j,2}$	$\epsilon(i,1)$
3	$1 \cdot 1$	$+\pi_{i,1}$	$\epsilon(j,2)$
4	$1 \cdot 1$	$+1$	$\epsilon((j,2),(i,1))$

Fig. 13. Relevant Terms for $\det \Delta_2(\epsilon)$ [39]

Implementations can precompute these coefficients and store them in a table. Figure 13 shows such a table for the quadratic matrix of size 2. Similar ones can be constructed for other dimensions ([39] also gives tables for dimensions 3 and 4). To compute the sign of a topological primitive, the implementation uses the table for a stepwise computation: The partial terms $\det M_t^{\Delta_2}$ are computed row by row. If the result is different is zero, the appropriate sign is returned.

Other perturbation schemes

The work of Emiris and Canny [43, 45] is based on the work of Edelsbrunner and Mücke. They improved the runtime efficiency by using other perturbations:

$$\begin{aligned}
\epsilon(i,j) &= \epsilon \cdot i^j \\
\epsilon(i,j) &= \epsilon \cdot (i^j \mod q) \\
\epsilon(i,j) &= \epsilon \cdot \tfrac{1}{i+j-1}
\end{aligned}$$

With them the handling of degenerate cases can be limited to a constant overhead.

In contrast to that, Yap [135, 136, 137, 138] proposed a different approach. It is more general and allows to perturb the input data in a controlled way. Thus, other perturbation schemes can

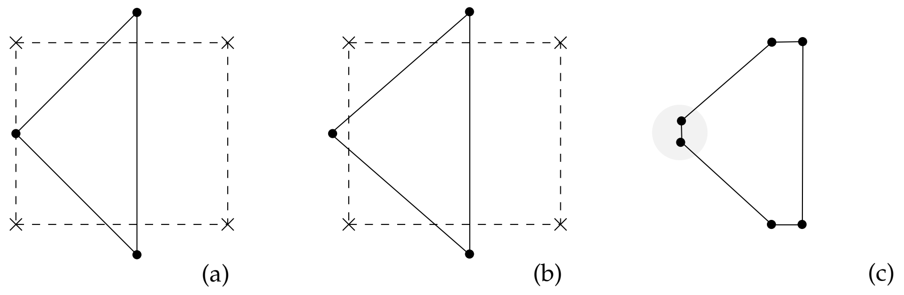

Fig. 14. Duplicated Vertices by Perturbation

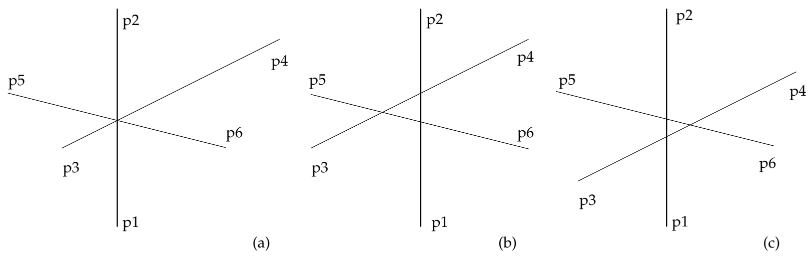

Fig. 15. Distinguishing Situations after Perturbations

be emulated so that they can be seen as special cases of Yap's scheme. However, the underlying mathematics is complicated, and the runtime is much worse than the ones of the other approaches presented above.

4.2.3 Critics

While symbolic perturbations are certainly a useful tool for the implementation of geometric algorithms, existing perturbation schemes have not shown to be as applicable as desired [16].

First, symbolic perturbations force the programmer a rather unsatisfactory choice: first, either to find an approximation of the original problem, or second, to find a precise solution of an approximation of the original problem. In some applications, both choices might be inappropriate, and a post-processing step is then required that determines the exact solution of the original problem. Besides its negative impact on the runtime, the complexity of the solution can be significantly increased.

To illustrates these drawbacks, consider the example shown in Figure 14. Although the perturbations are symbolic, the result is changed by them. If the intersection of both polygons of Figure 14 (a) is computed, the situation may be perturbed to the situation shown in part (b). The result polygon (c) contains two vertices with the same position. Without using perturbations, only one vertex would have been created. This difference must be corrected in an additional post-processing step.

Second, objects that are constructed by the algorithm (e. g. intersection points) are often forbidden in the computation, because their perturbation function must depend on the construction of the object. Hence, the new perturbation is either outside the definitions of primitive objects or it is much more complicated.

For instance, consider three line segments that intersect in a single point as shown in Figure 15. The perturbations of Edelsbrunner and Mücke (as well as the ones by Emiris and Canny) resolve the degeneracy of situation (a) by presenting either situation (b) or (c) to the user. However, the user has no chance to detect which situation has been chosen: If the algorithm needs to know whether the line segment between p_1 and p_2 is first crossed by the one between p_3 and p_4 or the one between p_5 and p_6, the perturbations are not usable. Equivalently, it cannot be decided whether the intersection point of the two crossing segments is above or below the first line segment. Actually, there is a solution for the problem presented in Figure 15, when seen in isolation. However, another incompatible perturbation must be used. So, it may be possible to compute a result, which is not consistent to the situation presented by the orientation predicate.

Third, symbolic perturbations need to be worked out in detail, a task that is generally very complex. This has been done only for a few geometric primitives. Moreover, since this task is very complex, its must be definitely verified if primitives based on it are used in safety-critical systems. This involves very complicated verification goals, which are based on a broad mathematical fundament. For instance, the formalisation depends on real numbers and many theorems that are currently not in the library of standard theorem provers. The necessary work for underlying mathematical theorems are out of all proportion to the verification results for this thesis.

Therefore, perturbation schemes do not play a role in the rest of the thesis. The polygon-processing library handles degenerate cases explicitly as the following section shows.

4.3 Three-Valued Logic

Three-valued logic has already proven to be useful in many areas, e. g. for the analysis of asynchronous circuits [14], in compiler analyses [116] or for causality analysis [5, 94, 118, 119, 124]. As it can be seen in the following, it allows one to describe geometric properties and algorithms more compactly and at the same time more precisely without having the need to enumerate many tedious cases. Clearly, degenerate cases do not disappear, but three-valued logic makes it possible to structure them and to treat them in a systematic and concise way.

Section 4.2.1 illustrated the problem of degenerated cases and showed that a reduction to two truth values is not an acceptable solution. Hence, for a degenerate case it is not decided to which basic case the situation is finally mapped. Instead, a third value is used to leave this case unspecified and to use this result for further computations. An intuitive view on the third truth value is obtained by the analogy to exception handling in some programming languages: At the point of time where an error occurs, it is not clear how to handle it. Thus, an exception is thrown, which is finally caught by a function that has the necessary knowledge to handle the error. Analogously, the degenerate case is passed through all functions until the knowledge of the context is sufficient to decide what to do.

As an example, consider the problem to check whether a point is on the edge, inside, or outside a polygon. Assume that the points on the edge are considered to be outside the polygon (i. e. polygons are ‘open’ sets of points). However, if the difference of two polygons is calculated by a set difference, the result is possibly a polygon that contains points on its edge. There are two ways to solve the problem: The first one is to modify the definition of the difference. The second one is not to decide whether points on the edge are inside or outside, and therefore using an undetermined, third value for these predicates.

	$\ddot{\neg}$
F	T
U	U
T	F

$\ddot{\wedge}$	F	U	T
F	F	F	F
U	F	U	U
T	F	U	T

$\ddot{\vee}$	F	U	T
F	F	U	T
U	U	U	T
T	T	T	T

$\ddot{\rightarrow}$	F	U	T
F	T	T	T
U	U	U	T
T	F	U	T

$\ddot{\leftrightarrow}$	F	U	T
F	T	U	F
U	U	U	U
T	F	U	T

$\ddot{\oplus}$	F	U	T
F	F	U	T
U	U	U	U
T	T	U	F

Fig. 16. Truth Tables of Basic Three-Valued Operators

$\ddot{*}$	F	U	T
F	F	F	F
U	F	U	T
T	F	T	T

$\twoheadrightarrow$	F	U	T
F	T	T	T
U	T	T	F
T	T	F	T

Fig. 17. Truth Tables of $\ddot{*}$ and $\twoheadrightarrow$

4.3.1 Definition of Three-Valued Logic

Thus, the area of a polygon is described by a function that maps each point of the plane to one of the three truth values: *true* (T) is assigned to all points inside, *false* (F) to all points outside, and *degenerate* (U) is assigned to all points on the edge of the polygon. These considerations give rise to the definitions of the basic three-valued connectives $\ddot{\neg}$, $\ddot{\wedge}$ and $\ddot{\vee}$ shown in Figure 16, which have already been used by Kleene [7, 84]. Further operators are defined (see Figure 16) like implication $\ddot{\rightarrow}$, equivalence $\ddot{\leftrightarrow}$ and exclusive-or $\ddot{\oplus}$.

$$\texttt{imp3} \vdash_{\text{def}} t_1 \ddot{\rightarrow} t_2 = \ddot{\neg} t_1 \ddot{\vee} t_2$$
$$\texttt{equ3} \vdash_{\text{def}} t_1 \ddot{\leftrightarrow} t_2 = t_1 \ddot{\wedge} t_2 \ \ddot{\vee} \ \neg t_1 \ddot{\wedge} \neg t_2$$
$$\texttt{xor3} \vdash_{\text{def}} t_1 \ddot{\oplus} t_2 = \neg t_1 \ddot{\wedge} t_2 \ \ddot{\vee} \ t_1 \ddot{\wedge} \neg t_2$$

The operators $\ddot{*}$ (a modified conjunction) and $\twoheadrightarrow$ (a modified implication) are well suited for the description of transitivity laws. This will be become clear in the following subsection. Their truth tables are given in Figure 17.
Moreover, the theory is extended by existential and universal quantification:

$$\begin{aligned}
&\texttt{exists3} \vdash_{\text{def}} \ddot{\exists} P = \\
&\qquad \textbf{if } (\exists x.Px = \mathsf{T}) \textbf{ then } \mathsf{T} \textbf{ else} \\
&\qquad\quad (\textbf{if } (\forall x.Px = \mathsf{F}) \textbf{ then } \mathsf{F} \textbf{ else } \mathsf{U}) \\
&\texttt{forall3} \vdash_{\text{def}} \ddot{\forall} P = \\
&\qquad \textbf{if } (\forall x.Px = \mathsf{T}) \textbf{ then } \mathsf{T} \textbf{ else} \\
&\qquad\quad (\textbf{if } (\exists x.Px = \mathsf{F}) \textbf{ then } \mathsf{F} \textbf{ else } \mathsf{U})
\end{aligned}$$

A closer inspection of the truth tables of the basic connectives $\ddot{\neg}$, $\ddot{\wedge}$, and $\ddot{\vee}$ reveals that these operations imply a natural ordering by the degree of truth: $\mathsf{F} < \mathsf{U} < \mathsf{T}$. In the context of this ordering, $\ddot{\neg}$ just reverses the values, $\ddot{\wedge}$ chooses the least one of its two operands and $\ddot{\vee}$ analogously the greatest one. Moreover, existential quantification $\ddot{\exists} x$ computes the maximum of a function $P : \mathcal{D}_x \to \mathbb{T}$, whereas universal quantification $\ddot{\forall} x$ computes the minimum. The relation $\ddot{\sqsubseteq} : \mathbb{T} \times$

$\ddot{\sqsubset}$	F	U	T
F	U	T	T
U	F	U	T
T	F	F	U

$\ddot{\sqsupset}$	F	U	T
F	U	F	F
U	T	U	F
T	T	T	U

$\ddot{\lhd}$	F	U	T
F	U	F	U
U	T	U	T
T	U	F	U

$\ddot{\rhd}$	F	U	T
F	U	T	U
U	F	U	F
T	U	T	U

Fig. 18. Truth Tables of $\ddot{\sqsubset}$, $\ddot{\sqsupset}$, $\ddot{\lhd}$ and $\ddot{\rhd}$

$\leq$	F	U	T
F	T	T	T
U	F	T	T
T	F	F	T

$\geq$	F	U	T
F	T	F	F
U	T	T	F
T	T	T	T

Fig. 19. Truth Tables of $\leq$ and $\geq$

$\mathbb{T} \to \mathbb{T}$ is defined that compares two truth values (see Figure 18). Consistently with the other operators, it will return U if both arguments are identical. The relation $\ddot{\sqsupset}$ is obtained by swapping the operands. Besides this ordering, there is yet another natural ordering which is given by the amount of knowledge: $\mathsf{U} < \mathsf{F}$ and $\mathsf{U} < \mathsf{T}$. Figure 18 gives the truth tables of $\ddot{\lhd}$ and $\ddot{\rhd}$.[1]

The two-valued theorem prover HOL is used to reason about the three-valued geometry predicates. Introducing three-valued formulas into such a two-valued environment poses the problem to integrate both logics. The conversion of three-valued expressions to Boolean domain depends on the proposition: In some situations, T should be the only designated truth value; in other cases, it suffices that a proposition P is 'at least U'. Although, this can be expressed by $\neg(P = \mathsf{F})$, two new relations $\leq$ and $\geq$ are introduced to improve the readability. By their help, all relevant cases ($P = \mathsf{F}$, $P \leq \mathsf{U}$, $P \geq \mathsf{U}$, $P = \mathsf{T}$) can be described concisely (see Figure 19).

In two-valued algorithms, conditionals are frequently used. The three-valued couterpart is the following definition. With its help, definitions can be made with a three-valued predicate as a guard.

$$\texttt{switch3} \vdash_{\text{def}} \mathsf{switch3}(\mathsf{t}, \mathsf{t}_1, \mathsf{t}_2, \mathsf{t}_3) =$$
$$\exists\, x.\, ((t = \mathsf{T}) \implies x = t_1) \wedge ((t = \mathsf{U}) \implies x = t_2) \wedge ((t = \mathsf{F}) \implies x = t_3)$$

4.3.2 Proof Tools

For proof obligations, special support is required in two areas: First, expressions involving three-valued logic should be handled efficiently. A collection of basic theorems and tactics that are natural extensions of the two-valued cases form in combination with an automatic reduction to two-valued expressions form a convenient tool set to cope with three-valued proof obligations, which will be presented in the following.

[1] Kleene postulated that the third truth value should be compatible with the increase in information. That means: if the value of some expression is changed from U to either T or F (gaining information), then the value of any formula containing this expression must not change from T to F or vice-versa. On the other hand, the formula may change from U to one of F or T (i.e. gaining information). Today, the regularity would be described in terms of monotonicity and lattices; assuming the order $\ddot{\lhd}$.

The system $\langle \mathbb{T}, \ddot{\vee}, \ddot{\wedge}, \ddot{\neg}, \mathsf{F}, \mathsf{T}, \mathsf{U} \rangle$ is a ternary algebra [14]: In addition to the laws of commutativity, associativity, distributivity, absorption and de Morgan as known from a Boolean algebra, the following theorems can be used for the transformation of three-valued terms:

`NOT_CLAUSES` $\vdash \ddot{\neg}\ddot{\neg}a = a \wedge (\ddot{\neg}\mathsf{U} = \mathsf{U})$
`AND_CLAUSES` $\vdash (\mathsf{T} \ddot{\wedge} t = t) \wedge (t \ddot{\wedge} \mathsf{T} = t) \wedge (\mathsf{F} \ddot{\wedge} t = \mathsf{F}) \wedge (t \ddot{\wedge} \mathsf{F} = \mathsf{F}) \wedge (t \ddot{\wedge} t = t)$
`OR_CLAUSES` $\vdash (\mathsf{F} \ddot{\vee} t = t) \wedge (t \ddot{\vee} \mathsf{F} = t) \wedge (\mathsf{T} \ddot{\vee} t = \mathsf{T}) \wedge (t \ddot{\vee} \mathsf{T} = \mathsf{T}) \wedge (t \ddot{\vee} t = t)$
`CONJ_COMM` $\vdash a \ddot{\wedge} b = b \ddot{\wedge} a$
`DISJ_COMM` $\vdash a \ddot{\vee} b = b \ddot{\vee} a$
`CONJ_ASSOC` $\vdash (a \ddot{\wedge} b) \ddot{\wedge} c = a \ddot{\wedge} b \ddot{\wedge} c$
`DISJ_ASSOC` $\vdash (a \ddot{\vee} b) \ddot{\vee} c = a \ddot{\vee} b \ddot{\vee} c$

`CONJ_ABSORP` $\vdash a \ddot{\wedge} (a \ddot{\vee} b) = a$
`DISJ_ABSORP` $\vdash a \ddot{\vee} a \ddot{\wedge} b = a$
`LEFT_AND_OVEROR` $\vdash a \ddot{\wedge} (b \ddot{\vee} c) = a \ddot{\wedge} b \ddot{\vee} a \ddot{\wedge} c$
`LEFT_OR_OVERAND` $\vdash a \ddot{\vee} b \ddot{\wedge} c = (a \ddot{\vee} b) \ddot{\wedge} (a \ddot{\vee} c)$
`RIGHT_AND_OVEROR` $\vdash (a \ddot{\vee} b) \ddot{\wedge} c = a \ddot{\wedge} c \ddot{\vee} b \ddot{\wedge} c$
`RIGHT_OR_OVERAND` $\vdash a \ddot{\wedge} b \ddot{\vee} c = (a \ddot{\vee} c) \ddot{\wedge} (b \ddot{\vee} c)$
`DE_MORGAN_THM` $\vdash (\ddot{\neg}(a \ddot{\wedge} b) = \ddot{\neg}a \ddot{\vee} \ddot{\neg}b) \wedge (\ddot{\neg}(a \ddot{\vee} b) = \ddot{\neg}a \ddot{\wedge} \ddot{\neg}b)$
`CONJ_TERNARY` $\vdash a \ddot{\wedge} \ddot{\neg}a \ddot{\wedge} \mathsf{U} = a \ddot{\wedge} \ddot{\neg}a$
`DISJ_TERNARY` $\vdash a \ddot{\vee} \ddot{\neg}a \ddot{\vee} \mathsf{U} = a \ddot{\vee} \ddot{\neg}a$

However, the complement laws of Boolean algebras are not satisfied:

`CONJ_BOOLCOMP` $\nvdash a \ddot{\wedge} \ddot{\neg}a = \mathsf{F}$
`DISJ_BOOLCOMP` $\nvdash a \ddot{\vee} \ddot{\neg}a = \mathsf{T}$

For interactive proofs, the theory offers several tactics that are adapted from the two-valued domain:

- `LOG3_GEN_TAC` strips the outermost universal quantifier from the conclusion of a goal. When applied to $A \vdash^{?} \ddot{\forall}x.P$, it reduces the goal to $A \vdash^{?} P[x'/x]$ where x' is a variant of x chosen to avoid clashing with any variables free in the assumption list of the goal. This tactic reduces both $\ddot{\forall}x.P(x) = \mathsf{T}$ and $\ddot{\exists}x.P(x) = \mathsf{F}$, since both express universal goals.
- `LOG3_EXISTS_TAC` reduces an existentially quantified goal to one involving a specific witness. When applied to a term u and a goal $\ddot{\exists}x.P$, `LOG3_EXISTS_TAC` reduces the goal to $P[u/x]$ (substituting u for all free instances of x in P, with variable renaming if necessary to avoid free variable capture).
- `LOG3_DISCH_TAC` moves the antecedent of a (three-valued) implicative goal into the assumptions.
- `LOG3_CONJ_TAC` reduces a conjunctive goal to two separate subgoals. When applied to a goal $A \vdash^{?} t_1 \ddot{\wedge} t_2$, the tactic reduces it to the two subgoals corresponding to each conjunct separately.
- `LOG3_DISJ1_TAC` and `LOG3_DISJ2_TAC` select the left and the right disjuncts of a disjunctive goal, respectively.
- `LOG3_EQ_TAC` reduces a goal of equivalence of three-valued terms to forward and backward implication. When applied to the goal $A \vdash^{?} t_1 \ddot{\leftrightarrow} t_2$, this tactic `EQ_TAC` returns the subgoals $A \vdash^{?} t_1 \ddot{\rightarrow} t_2$ and $A \vdash^{?} t_2 \ddot{\rightarrow} t_1$.

- Given a term u, `LOG3_CASES_TAC` applied to a goal produces three subgoals, one with $u = \mathsf{T}$ as an assumption, one with $u = \mathsf{U}$, and one with $u = \mathsf{F}$. A simple and very effective tactic to automatically prove simple theorems about the three-valued logic is `LOG3_EXPLORE_TAC`: It performs a case distinction on all free variables of the type $\mathbb{T}$ and then uses the simplifier.

A powerful tactic to prove goals specified in three-valued logic is the transformation to two-valued terms with a subsequent application of the traditional tactics for two-valued goals. For this purpose, a number of rewrite rules are provided that split up a three-valued proposition into positive atomic sub-proposition of the form $P = c$, $P \leq c$ or $P \geq c$ (where $c \in \{\mathsf{F}, \mathsf{U}, \mathsf{T}\}$) connected by two-valued operators. The complete reduction step is implemented by the tactic `LOG3_CALC_TAC` and involves the following steps:

- Elimination of non-constant expressions on the right hand side of equations and inequations:

$$\begin{aligned}
\texttt{LOG3\_CASES\_EQ} \vdash\ & (a = \mathsf{F}) \wedge (b = \mathsf{F}) \vee \\
& (a = \mathsf{U}) \wedge (b = \mathsf{U}) \vee \\
& (a = \mathsf{T}) \wedge (b = \mathsf{T}) \\
& = (a = b) \\
\texttt{LOG3\_CASES\_LEQ} \vdash\ & (a = \mathsf{F}) \wedge (b = \mathsf{F}) \vee \\
& (a \leq \mathsf{U}) \wedge (b = \mathsf{U}) \vee \\
& (b = \mathsf{T}) \\
& = a \leq b \\
\texttt{LOG3\_CASES\_GEQ} \vdash\ & (b = \mathsf{F}) \vee a \geq \mathsf{U} \wedge (b = \mathsf{U}) \vee \\
& (a = \mathsf{T}) \wedge (b = \mathsf{T}) \\
& = a \geq b
\end{aligned}$$

  In order to eliminate non-constant expressions on the right hand side, these rules must be applied from the right to the left. Of course, unconditional rewriting with these rules does not terminate.
- Elimination of propositions of the form $P = \mathsf{U}$: As the following theorems only consider the cases $P = \mathsf{F}$, $P = \mathsf{T}$, $P \leq \mathsf{U}$ and $P \geq \mathsf{U}$, rewriting (from right to left) with the following theorem eliminates propositions of the form $P = \mathsf{U}$.

$$\texttt{LOG3\_LEQ\_GEQ\_UU} \vdash a \leq \mathsf{U} \wedge a \geq \mathsf{U} = (a = \mathsf{U})$$

- Elimination of macro connectives: By rewriting with the definitions of $\ddot{\rightarrow}$, $\ddot{\leftrightarrow}$, $\ddot{\oplus}$ and $\ddot{\exists}$, all terms only consist of basic connectives.
- Elimination of basic connectives: All basic three-valued connectives can be reduced to two-valued connectives by the rewriting with theorems of the following form:

$$
\begin{aligned}
\texttt{LOG3\_NOT\_CALC} \vdash\ & ((\ddot{\neg} t = \mathsf{F}) = (t = \mathsf{T}))\wedge\\
& ((\ddot{\neg} t = \mathsf{T}) = (t = \mathsf{F}))\wedge\\
& (\ddot{\neg} t \le \mathsf{U} = t \ge \mathsf{U})\wedge\\
& (\ddot{\neg} t \ge \mathsf{U} = t \le \mathsf{U})\\
\texttt{LOG3\_AND\_CALC} \vdash\ & ((a \mathbin{\ddot{\wedge}} b = \mathsf{F}) = (a = \mathsf{F}) \vee (b = \mathsf{F}))\wedge\\
& ((a \mathbin{\ddot{\wedge}} b = \mathsf{T}) = (a = \mathsf{T}) \wedge (b = \mathsf{T}))\wedge\\
& ((a \mathbin{\ddot{\wedge}} b) \le \mathsf{U} = a \le \mathsf{U} \vee b \le \mathsf{U})\wedge\\
& ((a \mathbin{\ddot{\wedge}} b) \ge \mathsf{U} = a \ge \mathsf{U} \wedge b \ge \mathsf{U})\\
\texttt{LOG3\_OR\_CALC} \vdash\ & ((a \mathbin{\ddot{\vee}} b = \mathsf{F}) = (a = \mathsf{F}) \wedge (b = \mathsf{F}))\wedge\\
& ((a \mathbin{\ddot{\vee}} b = \mathsf{T}) = (a = \mathsf{T}) \vee (b = \mathsf{T}))\wedge\\
& ((a \mathbin{\ddot{\vee}} b) \le \mathsf{U} = a \le \mathsf{U} \wedge b \le \mathsf{U})\wedge\\
& ((a \mathbin{\ddot{\vee}} b) \ge \mathsf{U} = a \ge \mathsf{U} \vee b \ge \mathsf{U})\\
\texttt{LOG3\_EXT\_CALC} \vdash\ & ((\ddot{\triangle}\, a = \mathsf{F}) = \neg a)\wedge\\
& ((\ddot{\triangle}\, a = \mathsf{T}) = a)\wedge\\
& (\ddot{\triangle}\, a \le \mathsf{U} = \neg a)\wedge\\
& (\ddot{\triangle}\, a \ge \mathsf{U} = a)
\end{aligned}
$$

$$
\begin{aligned}
\texttt{LOG3\_EXISTS\_CALC} \vdash\ & ((\ddot{\exists} x.Px = \mathsf{F}) = \forall b.Pb = \mathsf{F})\wedge\\
& ((\ddot{\exists} x.Px = \mathsf{T}) = \exists b.Pb = \mathsf{T})\wedge\\
& ((\ddot{\exists} x.Px \le \mathsf{U}) = \forall b.Pb \le \mathsf{U})\wedge\\
& ((\ddot{\exists} x.Px \ge \mathsf{U}) = \exists b.Pb \ge \mathsf{U})\\
\texttt{LOG3\_FORALL\_CALC} \vdash\ & ((\ddot{\forall} x.P(x) = \mathsf{F}) = \exists b.P(b) = \mathsf{F})\wedge\\
& ((\ddot{\forall} x.P(x) = \mathsf{T}) = \forall b.P(b) = \mathsf{T})\wedge\\
& ((\ddot{\forall} x.P(x) \le \mathsf{U}) = \exists b.P(b) \le \mathsf{U})\wedge\\
& ((\ddot{\forall} x.P(x) \ge \mathsf{U}) = \forall b.P(b) \ge \mathsf{U})
\end{aligned}
$$

- Elimination of negative terms: All two-valued negations in front of subterms can be eliminated, leaving better understandable expressions.

$$
\begin{aligned}
\texttt{LOG3\_NOT2\_CALC} \vdash\ & (\neg(a = \mathsf{F}) = a \ge \mathsf{U})\wedge\\
& (\neg(a = \mathsf{T}) = a \le \mathsf{U})\wedge\\
& (\neg(a = \mathsf{U}) = (a = \mathsf{F}) \vee (a = \mathsf{T}))\wedge\\
& (\neg(a \le \mathsf{U}) = (a = \mathsf{T}))\wedge\\
& (\neg(a \ge \mathsf{U}) = (a = \mathsf{F}))\\
\texttt{LOG3\_ABS\_NOT} \vdash\ & (\ddot{\triangle}\, \neg a) = \ddot{\neg}(\ddot{\triangle}\, a)
\end{aligned}
$$

4.4 Three-Valued Primitives

Section 4.2 illustrated the problem of degenerated cases and showed that their explicit handling leads to an unmanageable number of cases computational geometry algorithms. Descriptions with three-valued operations reduce the complexity by consistently covering more cases with shorter descriptions as the following section shows.

4.4.1 Geometric Objects

Based on the analytic geometry theory and the three-valued logic, geometric objects and primitives can be defined. They are all formed by solution sets of points. To cope with endpoints and other extremal issues, three-valued inequations between rational numbers are used: For equal numbers, the inequation is neither false nor true, it is undefined. Hence, the three-valued less-than relation $r_1 \prec r_2$ over the rationals $\mathbb{Q}$ is defined as follows:

$$\texttt{les3} \vdash_{\text{def}} r_1 \prec r_2 = \\ \textbf{if } (r_1 < r_2) \textbf{ then } \mathsf{T} \textbf{ else } (\textbf{if } (r_2 < r_1) \textbf{ then } \mathsf{F} \textbf{ else } \mathsf{U})$$

Using this relation, geometric objects as sets of points can be defined in the following. A line is usually defined by its parametric equation. To convert the classic definition of a line to a three-valued one, all two-valued operators are exchanged by their three-valued counterparts:

$$\texttt{onLine} \vdash_{\text{def}} \mathsf{onLine}(l_1, v) = \\ \ddot{\exists}\lambda.\, (v = \mathsf{beg}(l_1) + \lambda \cdot \mathsf{dirvec}(l_1))$$

The term $\mathsf{dirvec}(l_1)$ denotes the direction vector of the line, which is the difference between the endpoint $\mathsf{end}(l_1)$ and the startpoint $\mathsf{beg}(l_1)$. For a line l, there is no difference between the two-valued and three-valued case: l contains all points $(x; y)$ that are a solution of the traditional, two-valued equation. For a ray and a line segment, the range of λ is restricted. Thus, the endpoints are degenerate cases:

$$\texttt{onRay} \vdash_{\text{def}} \mathsf{onRay}(l_1, v) = \\ \ddot{\exists}\lambda.\, (v = \mathsf{beg}(l_1) + \lambda \cdot (\mathsf{end}(l_1) - \mathsf{beg}(l_1)))\, \ddot{\wedge}\, (0 \prec \lambda)$$

$$\texttt{onSeg} \vdash_{\text{def}} \mathsf{onSegment}(l_1, v) = \\ \ddot{\exists}\lambda.\, (v = \mathsf{beg}(l_1) + \lambda \cdot (\mathsf{end}(l_1) - \mathsf{beg}(l_1)))\, \ddot{\wedge}\, (0 \prec \lambda \prec 1)$$

Two-dimensional objects can be defined analogously. A very important object is a rectangle that is aligned to the coordinate grid: If its lower, upper, left and right borders are given by $\mathsf{S}(w_1)$, $\mathsf{N}(w_1)$, $\mathsf{W}(w_1)$ and $\mathsf{E}(w_1)$ respectively, a point is inside this so-called window w_1 if it satisfies the following condition:

$$\texttt{inWindow} \vdash_{\text{def}} \mathsf{inWindow}(w_1, v) = \\ \mathsf{W}(w_1) \prec \mathsf{X}(v)\, \ddot{\wedge}\, \mathsf{X}(v) \prec \mathsf{E}(w_1)\, \ddot{\wedge}\, \mathsf{S}(w_1) \prec \mathsf{Y}(v)\, \ddot{\wedge}\, \mathsf{Y}(v) \prec \mathsf{N}(w_1)$$

4.4.2 Geometric Predicates

Degeneracies with respect to a predicate P are inputs x that cause the characteristic function to become zero $\chi_P(\mathbf{x}) = 0$. [2] Following the approach presented in Section 4.2, the result U is returned in these cases. Since all predicates defined at the beginning of the chapter compare their result with zero, the relation pos is additionally introduced:

$$\texttt{rat\_pos} \vdash_{\text{def}} \mathsf{pos}\,(r) = r \prec 0$$

With its help, the primitive $\mathsf{left}(v_1, v_2)$ and $\mathsf{below}(v_1, v_2)$ can be defined as follows:

$$\texttt{left} \vdash_{\text{def}} \mathsf{left}(v_1, v_2) = \mathsf{pos}\,(\mathsf{X}(v_2) - \mathsf{X}(v_1))$$
$$\texttt{right} \vdash_{\text{def}} \mathsf{right}(v_1, v_2) = \mathsf{left}(v_2, v_1)$$
$$\texttt{below} \vdash_{\text{def}} \mathsf{below}(v_1, v_2) = \mathsf{pos}\,(\mathsf{Y}(v_2) - \mathsf{Y}(v_1))$$
$$\texttt{above} \vdash_{\text{def}} \mathsf{above}(v_1, v_2) = \mathsf{below}(v_2, v_1)$$

[2] The fact that a primitive results to zero usually leads to problems in the layer below (analytic geometry), where these functions may be used in the denominator of an other definitions. For example, the line intersection has this problem for parallel lines.

The primitive left makes use of the three-valued relation $\prec$, and thus, it has similar properties: The following theorems prove some sort of reflexivity, antisymmetry, and transitivity laws.

$$\texttt{LEFT\_REF} \vdash \mathsf{left}(v_1, v_1) = \mathsf{U}$$
$$\texttt{LEFT\_ASYM} \vdash \mathsf{left}(v_1, v_2) = \ddot{\neg}\mathsf{left}(v_2, v_1)$$
$$\texttt{LEFT\_TRANS} \vdash \mathsf{left}(v_0, v_1) \ddot{*} \mathsf{left}(v_1, v_2) \twoheadrightarrow \mathsf{left}(v_0, v_2)$$

LEFT_TRANS makes use of the connectives $\ddot{*}$ and $\twoheadrightarrow$, which usually appear together in a proposition. They are especially well-suited for the description of transitive properties. For instance, the last theorem covers the following cases:

- If $\mathsf{left}(v_0, v_1) = \mathsf{T}$ and $\mathsf{left}(v_1, v_2) = \mathsf{T}$, then $\mathsf{left}(v_0, v_2) = \mathsf{T}$.
- If $\mathsf{left}(v_0, v_1) = \mathsf{T}$ and $\mathsf{left}(v_1, v_2) = \mathsf{U}$ or vice versa, then $\mathsf{left}(v_0, v_2) = \mathsf{T}$.
- If $\mathsf{left}(v_0, v_1) = \mathsf{U}$ and $\mathsf{left}(v_1, v_2) = \mathsf{U}$, then $\mathsf{left}(v_0, v_2) = \mathsf{U}$.
- If $\mathsf{left}(v_0, v_1) = \mathsf{F}$ or $\mathsf{left}(v_1, v_2) = \mathsf{F}$, then nothing is said about $\mathsf{left}(v_0, v_2)$.

The orientation primitives can be analogously defined:

$$\texttt{area3} \vdash_{\text{def}} \mathsf{triArea}(v_1, v_2, v_3) = (v_2 - v_1) \times (v_3 - v_2)$$
$$\texttt{lturn} \vdash_{\text{def}} \mathsf{lturn}(v_1, v_2, v_3) = \mathsf{pos}\ (\mathsf{triArea}(v_1, v_2, v_3))$$
$$\texttt{rturn} \vdash_{\text{def}} \mathsf{rturn}(v_1, v_2, v_3) = \mathsf{lturn}(v_3, v_2, v_1)$$

Again, various properties can be proven for the orientation primitive:

$$\texttt{LTURN\_REF} \vdash \mathsf{lturn}(v_1, v_1, v_2) = \mathsf{U}$$
$$\texttt{LTURN\_SYM} \vdash \mathsf{lturn}(v_1, v_2, v_3) = \mathsf{lturn}(v_2, v_3, v_1)$$
$$\texttt{LTURN\_ASYM} \vdash \mathsf{lturn}(v_1, v_2, v_3) = \ddot{\neg}(\mathsf{lturn}(v_2, v_1, v_3))$$

$$\begin{aligned}&\texttt{LTURN\_TRIAN} \vdash \\ &\quad \mathsf{lturn}(v_1, v_2, v_4) \ddot{*} \mathsf{lturn}(v_2, v_3, v_4) \ddot{*} \mathsf{lturn}(v_3, v_1, v_4) \twoheadrightarrow \\ &\quad \mathsf{lturn}(v_1, v_2, v_3)\end{aligned}$$
$$\begin{aligned}&\texttt{LTURN\_TRANS} \vdash \\ &\quad \mathsf{lturn}(v_1, v_2, v_3) \ddot{\wedge} \mathsf{lturn}(v_1, v_2, v_4) \ddot{\wedge} \mathsf{lturn}(v_1, v_2, v_5) \geq \mathsf{U} \implies \\ &\qquad \mathsf{lturn}(v_1, v_3, v_4) \ddot{*} \mathsf{lturn}(v_1, v_4, v_5) \twoheadrightarrow \mathsf{lturn}(v_1, v_3, v_5)\end{aligned}$$
$$\begin{aligned}&\texttt{LTURN\_MOD1} \vdash \\ &\quad \mathsf{onRay}(\mathsf{mkLine}(v_2, v_3), v_4) = \mathsf{T} \implies \\ &\qquad \mathsf{lturn}(v_1, v_2, v_3) = \mathsf{lturn}(v_1, v_2, v_4)\end{aligned}$$
$$\begin{aligned}&\texttt{LTURN\_MOD2} \vdash \\ &\quad \mathsf{onRay}(\mathsf{mkLine}(v_4, v_3), v_2) = \mathsf{T} \implies \\ &\qquad \mathsf{lturn}(v_1, v_2, v_4) = \mathsf{lturn}(v_1, v_3, v_4)\end{aligned}$$

These theorems are three-valued reformulation of the ones that can be found in [111]. The first three theorems (LTURN_REF, LTURN_SYM and LTURN_ASYM) state that a sequence in which a point appears at least twice is a degenerate case. Moreover, a sequence can be rotated without changing the orientation, and two points can be interchanged with negating the orientation of the sequence. LTURN_TRIAN describes the situation depicted in Figure 20 (a): If a point is on the positive side of three pairwise connected segments, they form a triangle with positive orientation. LTURN_TRANS proves the transitivity of the lturn predicate under the condition that the three points v_3, v_4 and v_5 lie on the positive side of a segment from v_1 to v_2 (see Figure 20 (b)). The last two theorems (Figure 20 (c) and (d)) are used in [111] to handle degenerate cases. Actually, they are not needed here, since LTURN_TRIAN already covers these cases. This illustrates the advantages of this

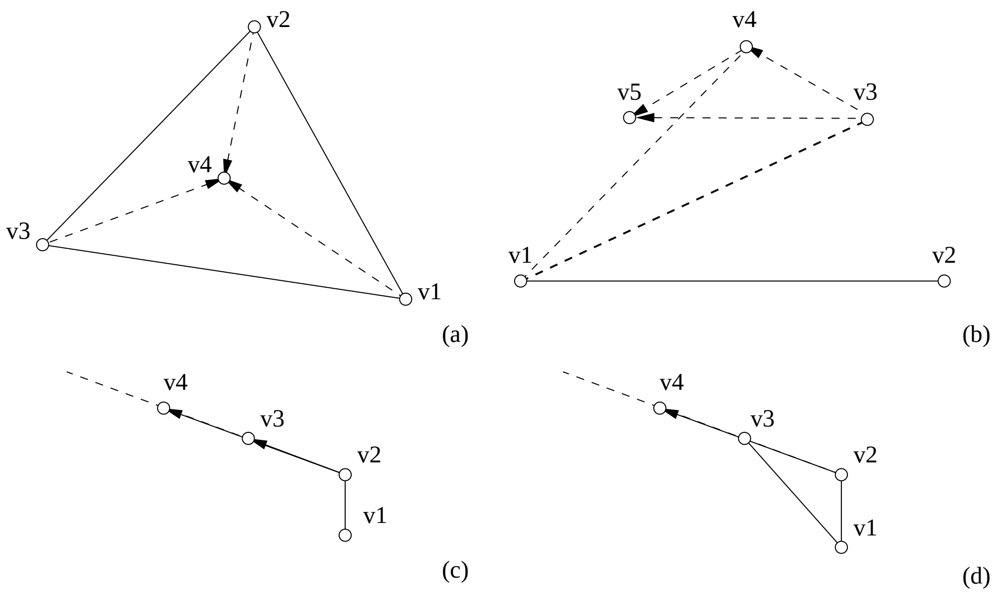

Fig. 20. Properties of the Left-Turn Predicate

approach: General and degenerate cases are always addressed at the same time, which makes the description succinct and readable. The same holds for software implementations that are made with three-valued data types.

4.4.3 Line Segement Intersection

This section gives some fundamental definitions and verifies some basic solutions, which can be found in various publications, e. g. in the article of Blinn [6].

An intersection primitive should test whether two given segments l_1 and l_2 intersect, and if so, the intersection point should be calculated. To address the problem of degenerate cases, i. e. touching and overlapping segments, three-valued predicates are used. Moreover, since the intersection can be again a segment, the type of the result of an intersection depends on the situation.

The existence of an intersection point is denoted by the following predicate doSegmentIntersect: Two line segments have a single intersection point if and only if the endpoints of one line are located on different sides of the other line. If doSegmentIntersect evaluates to U, the segments are either collinear or touch.

$$\texttt{left\_of\_line} \vdash_{\text{def}} \mathsf{leftOfLine}(l_1, v_1) = \mathsf{lturn}(\mathsf{beg}(l_1), \mathsf{end}(l_1), v_1)$$
$$\texttt{right\_of\_line} \vdash_{\text{def}} \mathsf{rightOfLine}(l_1, v_1) = \mathsf{rturn}(\mathsf{beg}(l_1), \mathsf{end}(l_1), v_1)$$
$$\begin{aligned}\texttt{do\_seg\_int} \vdash_{\text{def}} \mathsf{doSegmentIntersect}(l_1, l_2) = \\ (\mathsf{leftOfLine}(l_1, \mathsf{beg}(l_2)) \ddot{\leftrightarrow} \mathsf{rightOfLine}(l_1, \mathsf{end}(l_2))) \ddot{\wedge} \\ (\mathsf{leftOfLine}(l_2, \mathsf{beg}(l_1)) \ddot{\leftrightarrow} \mathsf{rightOfLine}(l_2, \mathsf{end}(l_1)))\end{aligned}$$

In practice, the existence of an intersection point can be often excluded before an actual check by the simple bounding box check. For two line segments l_1 and l_2, the predicate $\mathsf{bboxCheck}(l_1, l_2)$ must be true if an intersection point exists. In the case of U, they can only touch.

$$\begin{aligned}
&\texttt{vecXint} \vdash_{\text{def}} \mathsf{vecXint}(v_1, v_2, v_3) = (\mathsf{X}(v_1) \prec \mathsf{X}(v_2)) \ddot{\wedge} (\mathsf{X}(v_2) \prec \mathsf{X}(v_3))\\
&\texttt{vecYint} \vdash_{\text{def}} \mathsf{vecYint}(v_1, v_2, v_3) = (\mathsf{X}(v_1) \prec \mathsf{X}(v_2)) \ddot{\wedge} (\mathsf{X}(v_2) \prec \mathsf{X}(v_3))\\
&\texttt{bbox\_check} \vdash_{\text{def}} \mathsf{bboxCheck}(l_1, l_2) =\\
&\quad (\mathsf{vecXint}(\mathsf{beg}(l_1), \mathsf{beg}(l_2), \mathsf{end}(l_1)) \ddot{\vee} \mathsf{vecXint}(\mathsf{beg}(l_1), \mathsf{end}(l_2), \mathsf{end}(l_1)) \ddot{\vee}\\
&\qquad \mathsf{vecXint}(\mathsf{beg}(l_2), \mathsf{beg}(l_1), \mathsf{end}(l_2)) \ddot{\vee} \mathsf{vecXint}(\mathsf{beg}(l_2), \mathsf{end}(l_1), \mathsf{end}(l_2))\ ) \ddot{\wedge}\\
&\quad (\mathsf{vecYint}(\mathsf{beg}(l_1), \mathsf{beg}(l_2), \mathsf{end}(l_1)) \ddot{\vee} \mathsf{vecYint}(\mathsf{beg}(l_1), \mathsf{end}(l_2), \mathsf{end}(l_1)) \ddot{\vee}\\
&\qquad \mathsf{vecYint}(\mathsf{beg}(l_2), \mathsf{beg}(l_1), \mathsf{end}(l_2)) \ddot{\vee} \mathsf{vecYint}(\mathsf{beg}(l_2), \mathsf{end}(l_1), \mathsf{end}(l_2))\ )
\end{aligned}$$

If an intersection point exists, it is calculated by the following function:

$$\begin{aligned}
&\texttt{vec\_on\_line} \vdash_{\text{def}} \mathsf{vecAt}\,(l_1, r_1) = \mathsf{beg}(l_1) + r_1 \cdot \mathsf{dirvec}(l_1)\\
&\texttt{line\_intersect} \vdash_{\text{def}}\\
&\quad \mathsf{lineIntersect}(l_1, l_2) = \mathsf{vecAt}\left(l_2, \frac{(\mathsf{beg}(l_1) - \mathsf{beg}(l_2)) \times \mathsf{dirvec}(l_1)}{\mathsf{dirvec}(l_2) \times \mathsf{dirvec}(l_1)}\right)
\end{aligned}$$

In order to see that these definitions make sense, they are integrated in the theorem prover framework. The intersection point of two line segments l_1 and l_2 should be on both lines.

$$\begin{aligned}
&\texttt{exists\_seg\_int} \vdash_{\text{def}} \mathsf{existsSegmentIntersect}(l_1, l_2) = \ddot{\exists} v_1.\mathsf{isSegmentIntersect}(l_1, l_2, v_1)\\
&\texttt{is\_seg\_int} \vdash_{\text{def}} \mathsf{isSegmentIntersect}(l_1, l_2, v_1) = \mathsf{onSegment}(l_1, v_1) \ddot{\wedge} \mathsf{onSegment}(l_2, v_1)
\end{aligned}$$

Based on the results of Section 4.1.2, the definitions can be checked now:

$$\begin{aligned}
&\texttt{SEGINT\_CALC\_CORRECT} \vdash\\
&\quad \mathsf{leftOfLine}(l_1, \mathsf{beg}(l_2)) \ddot{\leftrightarrow} \mathsf{rightOfLine}(l_1, \mathsf{end}(l_2)) \ddot{\wedge}\\
&\qquad \mathsf{leftOfLine}(l_2, \mathsf{beg}(l_1)) \ddot{\leftrightarrow} \mathsf{rightOfLine}(l_2, \mathsf{end}(l_1)) \implies\\
&\qquad\quad (\forall v.(\mathsf{isSegmentIntersect}(l_1, l_2, v) = \mathsf{T}) = (v = \mathsf{lineIntersect}(l_1, l_2)))\\
&\texttt{BBOX\_CHECK\_CORRECT} \vdash \mathsf{parallel}(l_1, l_2)\\
&\quad (\mathsf{bboxCheck}(l_1, l_2) \ddot{\rightarrow} \mathsf{existsSegmentIntersect}(l_1, l_2) = \mathsf{T})
\end{aligned}$$

In the proof of the first theorem, `LINEINT_CALC_CORRECT` can be used to substitute the expression with $\mathsf{lineIntersect}$ by $\mathsf{isLineIntersect}$. Now, the backward implication of the equation is obvious. It remains to prove that the 'crossing' property of the segments implies that the parameter λ of the parametric forms of both segments remains between 0 and 1. This can be done by rewriting and by using the order properties of the rational numbers (see Section 2.2). Analogously, the bounding box check is verified.

Overlay

Another important definition is the overlay of two line segments. In contrast to the intersection of two lines, it covers the case of overlapping line segments. The overlay of two line segments is a set of segments with the following two properties: First, the resulting segments neither intersect nor overlap — they may only contain common *endpoints*. Second, they cover the two input segments. Third, no point not belonging to the two given segments belongs to the overlay:

```
conditional_line $\vdash_{\text{def}}$ $\mathsf{optLine}(\mathsf{v}_1, \mathsf{v}_2) =$
    if $v_1 \neq v_2$ then $\{\mathsf{mkLine}(v_1, v_2)\}$ else $\{\}$
conditional_intersect $\vdash_{\text{def}}$ $\mathsf{optIntersect}(\mathsf{l}_1, \mathsf{l}_2) =$
    if $\mathsf{segDoIntersect}(l_1, l_2) = \mathsf{T}$ then $\{\mathsf{lineIntersect}(l_1, l_2)\}$ else $\{\}$
seg_overlay $\vdash_{\text{def}}$ $\mathsf{segOverlayLines}(\mathsf{l}_1, \mathsf{l}_2) =$
    if $\mathsf{segDoOverlap}(l_1, l_2) = \mathsf{T}$ then
      let $\langle v_1, v_2, v_3, v_4\rangle = \mathsf{flatLexOrder}(\{\mathsf{beg}(l_1), \mathsf{end}(l_1), \mathsf{beg}(l_2), \mathsf{end}(l_2)\})$ in
        $\mathsf{optLine}(\mathsf{v}_1, \mathsf{v}_2) \cup \mathsf{optLine}(\mathsf{v}_2, \mathsf{v}_3) \cup \mathsf{optLine}(\mathsf{v}_3, \mathsf{v}_4)$
    else if $\mathsf{segDoIntersect}(l_1, l_2) \geq \mathsf{U}$ then
      let $v = \mathsf{lineIntersect}(l_1, l_2)$ in
        $\mathsf{optLine}(\mathsf{beg}(\mathsf{l}_1), \mathsf{v}) \cup \mathsf{optLine}(\mathsf{beg}(\mathsf{l}_2), \mathsf{v}) \cup \mathsf{optLine}(\mathsf{v}, \mathsf{end}(\mathsf{e}_1)) \cup \mathsf{optLine}(\mathsf{v}, \mathsf{end}(\mathsf{l}_2))$
    else $\{l_1, l_2\}$
seg_overlay_points $\vdash_{\text{def}}$ $\mathsf{segOverlayPoints}(\mathsf{l}_1, \mathsf{l}_2) =$
      $\{\mathsf{beg}(l_1), \mathsf{end}(l_1), \mathsf{beg}(l_2), \mathsf{end}(l_2)\} \cup \mathsf{optIntersect}(\mathsf{l}_1, \mathsf{l}_2)$
```

The definitions above satisfy the intended conditions as the following theorems assert.

```
SEG_OVERLAY_CORRECT $\vdash$
    $l_0 \in \mathsf{segOverlayLines}(\mathsf{l}_1, \mathsf{l}_2) \implies$
    $\mathsf{onSegment}(l_0, v) \leq (\mathsf{onSegment}(l_1, v) \ddot{\vee} \mathsf{onSegment}(l_2, v))$
SEG_OVERLAY_COMPLETE $\vdash$
    $l_0 \in \mathsf{segOverlayLines}(\mathsf{l}_1, \mathsf{l}_2) \implies$
    $\mathsf{onSegment}(l_0, v) \geq \mathsf{U} \implies$
    $\mathsf{onSegment}(l_1, v) \ddot{\vee} \mathsf{onSegment}(l_2, v) \geq \mathsf{U}$
```

4.4.4 Convex Hull Algorithm

As an example to illustrate the approach with three-valued primitives, consider a convex hull algorithm. The convex hull of a set of points is a subset that, when connected, contains all other input points.

The formalisation follows the implementation presented in [36]. It divides the computation of the convex hull into two symmetric parts: the upper part and the lower part of the hull (see Figure 21 (a)). In this section, the focus is on the construction of the lower part. The upper part is handled absolutely analogously.

Formalisation

The algorithm takes a list of points $\mathcal{L}$, which is sorted lexicographically (denoted as $\mathsf{lexSorted}(\mathcal{L})$), i. e. points are first sorted by their x-coordinates and then by their y-coordinates. The points are iteratively added to the lower part of the convex hull. After each addition, it is checked whether the last three points form a left turn. If this is not the case, the middle point is deleted. These steps are repeated until the last three points make a left turn, or there are only two points left (the leftmost point and the added point). Figure 21 (b) illustrates this procedure. Formally, the construction of the lower hull can be described by the following functions, where $\langle\rangle$ denotes the empty list, $\langle e_1; e_2\rangle$ a list containing the two elements e_1 and e_2, and $e :: \mathcal{L}$ denotes the concatenation of a new leftmost element e to an existing list $\mathcal{L}$:

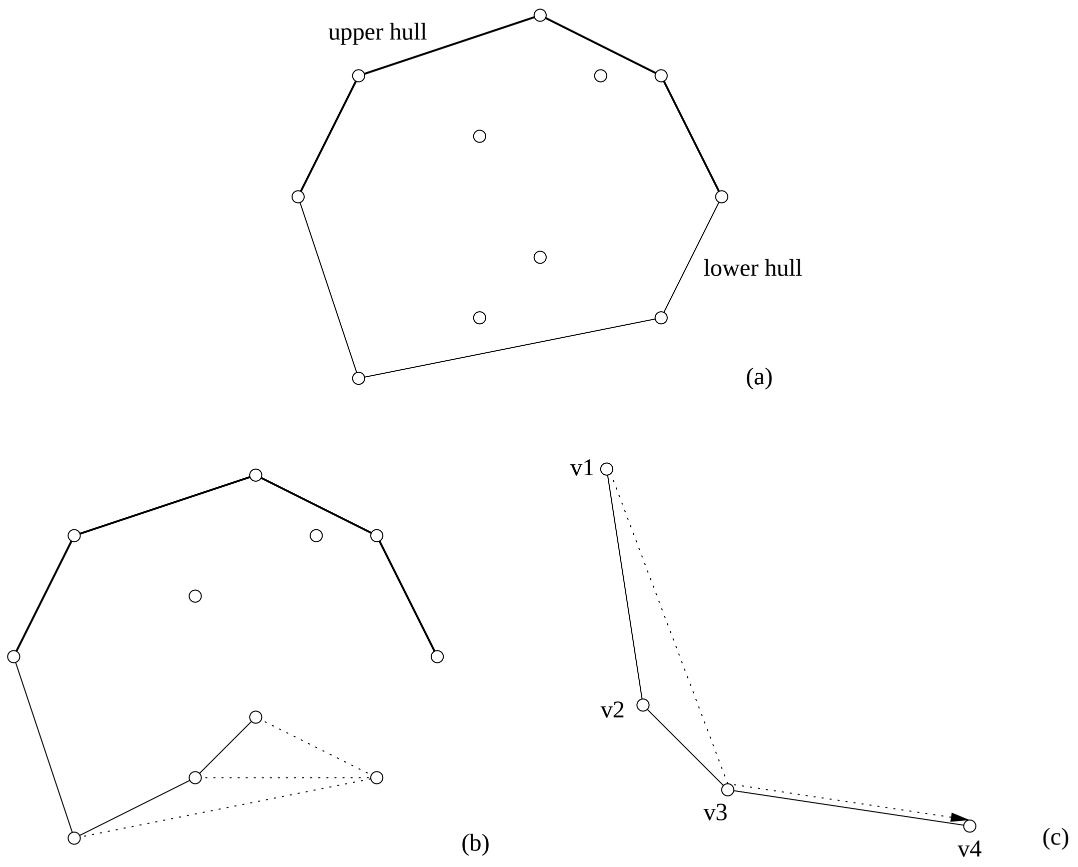

Fig. 21. Computation of the Convex Hull

```
normalize_lower $\vdash_{\text{def}}$
    $(\mathsf{NormLow}(\langle\rangle) = \langle\rangle)\wedge$
    $(\mathsf{NormLow}(\langle e_1\rangle) = \langle e_1\rangle)\wedge$
    $(\mathsf{NormLow}(\langle e_1; e_2\rangle) = \langle e_1; e_2\rangle)\wedge$
    $(\mathsf{NormLow}((e_1 :: e_2 :: e_3 :: \mathcal{L})) =$
        **if** $\mathsf{lturn}(e_1, e_2, e_3) = \mathsf{T}$ **then** $e_1 :: e_2 :: e_3 :: \mathcal{L}$
        **else** $\mathsf{NormLow}((e_1 :: e_3 :: \mathcal{L})))$
hull_lower $\vdash_{\text{def}}$
    $(\mathsf{LowerHull}(\mathcal{L}_0, \langle\rangle) = \mathcal{L}_0)\wedge$
    $(\mathsf{LowerHull}(\mathcal{L}_0, e :: \mathcal{L}_1) = \mathsf{NormLow}(\mathsf{LowerHull}(e :: \mathcal{L}_0, \mathcal{L}_1)))$
```

If $\mathcal{L}$ has at least three elements, $\mathsf{NormLow}(\mathcal{L})$ deletes the second element if the first three elements should not form a left turn, and hullLow applies this function to all sublists of a list $\mathcal{L}$.

Specification

A sequence of points is part of the convex hull if for two consecutive points, all other points lie on the left hand side of the line between these points. The corresponding predicate leftConvex is defined recursively: A sequence of no elements or one element is always convex. Each additional point that is added must lie on the left of all former edges of the constructed convex hull (leftPoint), and all points must lie on the left side of the edge that is created by the insertion of the new point (leftEdge).

$$
\begin{array}{l}
\texttt{left\_edge} \vdash_{\text{def}} \\
\quad (\mathsf{leftEdge}(e_1, e_2, [\,]))\wedge \\
\quad (\mathsf{leftEdge}(e_1, e_2, e :: \mathcal{L}) = \\
\qquad (\mathsf{lturn}(e, e_1, e_2) = \mathsf{T}) \wedge \mathsf{leftEdge}(e_1, e_2, \mathcal{L})) \\
\texttt{left\_point} \vdash_{\text{def}} \\
\quad (\mathsf{leftPoint}(e, [\,]))\wedge \\
\quad (\mathsf{leftPoint}(e, [e_1]))\wedge \\
\quad (\mathsf{leftPoint}(e, e_1 :: e_2 :: \mathcal{L}) = \\
\qquad (\mathsf{lturn}(e, e_2, e_1) = \mathsf{T}) \wedge \mathsf{leftPoint}(e, e_2 :: \mathcal{L})) \\
\texttt{left\_convex} \vdash_{\text{def}} \\
\quad (\mathsf{leftConvex}([\,]))\wedge \\
\quad (\mathsf{leftConvex}([e_1]))\wedge \\
\quad (\mathsf{leftConvex}(e_1 :: e_2 :: \mathcal{L}) = \\
\qquad \mathsf{leftEdge}(e_1, e_2, \mathcal{L}) \wedge \mathsf{leftPoint}(e_1, e_2 :: \mathcal{L}) \wedge \mathsf{leftConvex}(e_2 :: \mathcal{L}))
\end{array}
$$

Verification

The verification is done in several steps. First, by applying the definitions, it is proven that every sublist of three points in the result makes a left turn.

$$
\begin{array}{l}
\texttt{left\_chain} \vdash_{\text{def}} \\
\quad (\mathsf{leftChain}([\,]))\wedge \\
\quad (\mathsf{leftChain}([e_1]))\wedge \\
\quad (\mathsf{leftChain}([e_1; e_2]))\wedge \\
\quad (\mathsf{leftChain}(e_1 :: e_2 :: e_3 :: \mathcal{L}) = \\
\qquad (\mathsf{lturn}(e_1, e_2, e_3) = \mathsf{T}) \wedge \mathsf{leftChain}(e_2 :: e_3 :: \mathcal{L})) \\
\texttt{LEFT\_CHAIN\_HULL\_LOWER} \vdash \\
\quad \mathsf{leftChain}(\mathcal{L}_0) \Rightarrow \mathsf{leftChain}(\mathsf{LowerHull}(\mathcal{L}_0, \mathcal{L}_1))
\end{array}
$$

Then, under the condition of a lexicographic ordering a kind of transitivity (see Figure 21 (c)) is derived. To prove this, the lexicographic conditions are translated to left turn conditions before the transitivity of the left-turn predicate `LTURN_TRANS` is used. With the help of this lemma, an induction proves the desired theorem `CVX_LOWER`.

$$
\begin{array}{l}
\texttt{CVX\_TRANS\_LOWER} \vdash \\
\quad (\mathsf{lturn}(v_1, v_2, v_3) = \mathsf{T}) \wedge (\mathsf{lturn}(v_2, v_3, v_4) = \mathsf{T})\wedge \\
\quad (v_1 \prec v_2 = \mathsf{T}) \wedge (v_2 \prec v_3 = \mathsf{T}) \wedge (v_3 \prec v_4 = \mathsf{T}) \\
\quad \Rightarrow (\mathsf{lturn}(v_1, v_3, v_4) = \mathsf{T}) \\
\texttt{CVX\_LOWER} \vdash \\
\quad \mathsf{lexSorted}(\mathcal{L}) \wedge \mathsf{leftChain}(\mathcal{L}) \Rightarrow \mathsf{leftConvex}(\mathcal{L})
\end{array}
$$

Note that in the proofs, degenerate cases do not have to be addressed explicitly. Instead, it is exploited that theorems like `LTURN_TRANS` subsume many cases. Thus, the correctness of the algorithm is guaranteed for all cases: in particular for the situation that two subsequent input points have the same y-coordinate or there are collinear points in the input set.

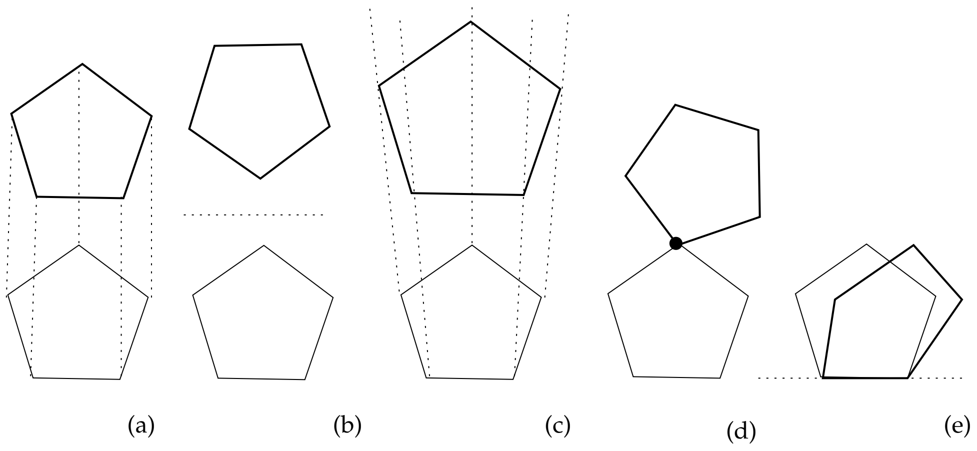

Fig. 22. Affine Transformations

4.5 Symmetries

Very often, various situations in geometry seem to be very similar. For instance, it is easy to see that for all the theorems defined with the help of the left-turn predicate analogous theorems can be defined for the right-turn predicate. Similarly, a point remains above (or below) another one if the whole plane is mirrored at the y-axis. Since the underlying problems are more or less equivalent, some transformations of the geometric situation do not really change a problem. In particular for the verification tasks, it is very tedious to prove theorems that have been proven in a similar way before. In contrast to textual proof descriptions, where similar situations are usually skipped by stating that a proof is analogous to another one, the theorem prover must be given a formal definition what *similar* means and where it can apply *analogous* proof techniques.

4.5.1 Transformations

To this end, basic transformations are defined. For linear geometry, affine transformations form such an important class. Figure 22 illustrates the operations that belong to this class: translation (a), mirroring (b), scaling (c), rotating (d) and shearing (e). In the following, not exactly this group of operations but a slightly modified set of operations is used: Mirroring and rotating will be restricted aligned to the coordinate system, and shearing will not be considered at all. Instead, scaling can be done coordinate-wise. These transformations have a practical relevance for the polygon-processing library, even though all others can be used in principle for the symmetry approach. Thus, the following primitive transformations are used, which can be combined in later proofs:

$\texttt{translate} \vdash_{\text{def}} \mathsf{trans}(v, t) = v + t$
$\texttt{xmirror} \vdash_{\text{def}} \mathsf{xmir}(v) = (-\mathsf{X}(v); \mathsf{Y}(v))$
$\texttt{ymirror} \vdash_{\text{def}} \mathsf{ymir}(v) = (\mathsf{X}(v); -\mathsf{Y}(v))$
$\texttt{xscale} \vdash_{\text{def}} \mathsf{xscale}(v, r) = (r \cdot \mathsf{X}(v); \mathsf{Y}(v))$
$\texttt{yscale} \vdash_{\text{def}} \mathsf{yscale}(v, r) = (\mathsf{X}(v); r \cdot \mathsf{Y}(v))$
$\texttt{cwrotate} \vdash_{\text{def}} \mathsf{cw}(v) = \mathsf{orth}(v)$
$\texttt{ccwrotate} \vdash_{\text{def}} \mathsf{ccw}(v) = \mathsf{orth}(-v)$

With the theorem prover, it is now analysed what properties can be derived for all the standard predicates defined in this chapter. For instance, for the left-turn predicate, the following theorems have been proven:

$$\begin{array}{l}
\texttt{LTURN\_TRANSLATE} \vdash \mathsf{lturn}(\mathsf{trans}(v_1, t), \mathsf{trans}(v_2, t), \mathsf{trans}(v_3, t)) = \mathsf{lturn}(v_1, v_2, v_3) \\
\texttt{LTURN\_XMIRROR} \vdash \mathsf{lturn}(\mathsf{xmir}(v_1), \mathsf{xmir}(v_2), \mathsf{xmir}(v_3)) = \mathsf{rturn}(v_1, v_2, v_3) \\
\texttt{LTURN\_YMIRROR} \vdash \mathsf{lturn}(\mathsf{ymir}(v_1), \mathsf{ymir}(v_2), \mathsf{ymir}(v_3)) = \mathsf{rturn}(v_1, v_2, v_3) \\
\texttt{LTURN\_XSCALE} \vdash (r \neq 0) \implies \\
\quad (\mathsf{lturn}(\mathsf{xscale}(v_1, r), \mathsf{xscale}(v_2, r), \mathsf{xscale}(v_3, r)) = \mathsf{lturn}(v_1, v_2, v_3)) \\
\texttt{LTURN\_YSCALE} \vdash (r \neq 0) \implies \\
\quad (\mathsf{lturn}(\mathsf{yscale}(v_1, r), \mathsf{yscale}(v_2, r), \mathsf{yscale}(v_3, r)) = \mathsf{lturn}(v_1, v_2, v_3)) \\
\texttt{LTURN\_CWROTATE} \vdash \mathsf{lturn}(\mathsf{cw}(v_1), \mathsf{cw}(v_2), \mathsf{cw}(v_3)) = \mathsf{lturn}(v_1, v_2, v_3) \\
\texttt{LTURN\_CCWROTATE} \vdash \mathsf{lturn}(\mathsf{ccw}(v_1), \mathsf{ccw}(v_2), \mathsf{ccw}(v_3)) = \mathsf{lturn}(v_1, v_2, v_3)
\end{array}$$

Analogous theorems can be derived for all other predicates. To raise the abstraction level, transformations are not only defined on vectors, but also on composed objects like lines and windows. For example, the translation is defined as follows:

$$\begin{array}{l}
\texttt{translate\_line} \vdash_{\mathrm{def}} \mathsf{lineTrans}(l_1, t) = \mathsf{mkLine}(\mathsf{trans}(\mathsf{beg}(l_1), t), \mathsf{trans}(\mathsf{end}(l_1), t)) \\
\texttt{translate\_window} \vdash_{\mathrm{def}} \mathsf{winTrans}(l_1, t) = \mathsf{mkWin}(\mathsf{trans}(\mathsf{SW}(l_1), t), \mathsf{trans}(\mathsf{NE}(l_1), t))
\end{array}$$

4.5.2 Cohen-Sutherland Line Clipping

The Cohen-Sutherland example demonstrates how the primitives can be used to verify a simple algorithm from computer graphics. It shows how three-valued primitives are finally mapped to two-valued primitives in the context of a concrete application, and how symmetries can be used to handle many similar goals.

Description

A frequent operation in many graphic applications is the line clipping to an upright rectangular window. To this end, the Cohen-Sutherland algorithm divides the plane into nine regions: Each of the edges of the clip window defines an infinite line that divides the plane into inside and outside half-spaces (see Figure 23 (a)). The resulting regions can be described using a four bit *outcode*. The four bits of this code denote whether this region is situated above (*north*), below (*south*), left (*west*) and right (*east*) of the window, respectively.

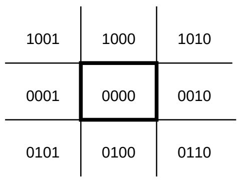

Fig. 23. Cohen-Sutherland: *Outcodes*

With the help of the outcodes, necessary conditions can be formulated to check whether the line segment is inside or outside the window: The line segment is inside the window if and only if no bit of the outcodes of the endpoints is set. On the other side, if a common bit in each outcode of the endpoints is set, then both endpoints are outside the same side of the window, and thus, the entire line segment is outside the window. In most cases, a line segment will have been either accepted or rejected by these conditions.

In all other cases, the line segment is split into two pieces at one of the four clipping edges. Assume that the considered endpoint $(x_1; y_1)$ is above the window w (see Figure 24 (b)). Removing the portion of the line that is above the window results in a new line segment with the old endpoint $(x_2; y_2)$ and the new endpoint $(x'_1; y'_1)$. Since the new endpoint is on the top border of the window, $y'_1 = \mathsf{N}(w)$ holds. The other coordinate x'_1 can be computed as follows:

$$x'_1 = x_1 + (x_2 - x_1) \cdot \frac{\mathsf{N}(w) - y_1}{y_2 - y_1}$$

Once the line segment is identified, the outcode of the new endpoint is computed. After this, the algorithm is restarted with the new values.

Formalisation

The formalisation of the algorithm in HOL starts with the definition of the outcodes. Let $\mathsf{SW}(w)$ be the lower left corner of w and $\mathsf{NE}(w)$ the upper right corner of the window w, respectively. These corners are just the components of the underlying representation.

`outcode` $\vdash_{\text{def}}$

$$\mathsf{Outcode}(w, v) = \begin{pmatrix} \mathsf{below}(v, \mathsf{SW}(w)) = \mathsf{T}, \\ \mathsf{above}(v, \mathsf{NE}(w)) = \mathsf{T}, \\ \mathsf{left}(v, \mathsf{SW}(w)) = \mathsf{T}, \\ \mathsf{right}(v, \mathsf{NE}(w)) = \mathsf{T} \end{pmatrix}$$

Based on this, the outcode predicates are defined for an arbitrary outcode c as follows:

`INSIDE` $\vdash_{\text{def}} \mathsf{Inside}(c_1) = (c_1 = (\mathsf{F}, \mathsf{F}, \mathsf{F}, \mathsf{F}))$

`BOTTOM` $\vdash_{\text{def}} \mathsf{South}(c_1) = c_1[0]$

`TOP` $\vdash_{\text{def}} \mathsf{North}(c_1) = c_1[1]$

`LEFT` $\vdash_{\text{def}} \mathsf{West}(c_1) = c_1[2]$

`RIGHT` $\vdash_{\text{def}} \mathsf{East}(c_1) = c_1[3]$

`ACCEPT` $\vdash_{\text{def}} \mathsf{Accept}(c_1, c_2) = \mathsf{Inside}(c_1) \wedge \mathsf{Inside}(c_2)$

`REJECT` $\vdash_{\text{def}}$

$$\mathsf{Reject}(c_1, c_2) = \begin{pmatrix} c_1[0] \wedge c_2[0] \ \vee \ c_1[1] \wedge c_2[1] \vee \\ c_1[2] \wedge c_2[2] \ \vee \ c_1[3] \wedge c_2[3] \end{pmatrix}$$

The actual algorithm is taken from [49], where the loop is replaced by a recursive call. Moreover, as the algorithm does not return an edge in all cases, an *option type* is used for the result, returning None if the line is rejected, and $\mathsf{Some}(e)$ if the edge e is accepted.

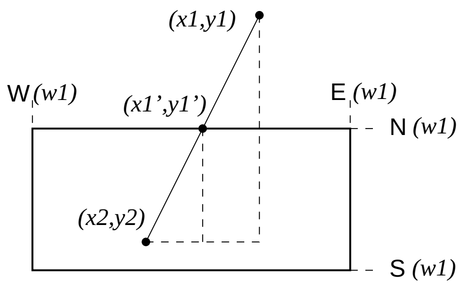

Fig. 24. Cohen-Sutherland: computing the intersection

```
csa_clip $\vdash_{\text{def}}$ $\mathsf{CSAClip}(w, (v_b, v_e)) =$
    if $\mathsf{Accept}(\mathsf{Outcode}(w, v_b), \mathsf{Outcode}(w, v_e))$ then $\mathsf{Some}((v_b, v_e))$
     else if $\mathsf{Reject}(\mathsf{Outcode}(w, v_b), \mathsf{Outcode}(w, v_e))$ then $\mathsf{None}$
     else if $\mathsf{Inside}(\mathsf{Outcode}(w, v_b))$ then $\mathsf{CSAClip}(w, (v_e, v_b))$
     else $\mathsf{CSAClip}(w, \mathsf{ShortenedLine}(w, (v_b, v_e)))$

shortened_line $\vdash_{\text{def}}$ $\mathsf{ShortenedLine}(w, (v_b, v_e)) =$
    let
      $x_1 = \mathsf{X}(v_b)$ and $y_1 = \mathsf{Y}(v_b)$ and
      $x_2 = \mathsf{X}(v_e)$ and $y_2 = \mathsf{Y}(v_e)$ and
      $c_1 = \mathsf{Outcode}(w, v_b)$
    in
      if $\mathsf{North}(c_1)$ then $\left(\left(x_1 + (x_2 - x_1) \cdot \frac{\mathsf{N}(w) - y_1}{y_2 - y_1}; \mathsf{N}(w)\right), v_e\right)$
       else if $\mathsf{South}(c_1)$ then $\left(\left(x_1 + (x_2 - x_1) \cdot \frac{\mathsf{S}(w) - y_1}{y_2 - y_1}; \mathsf{S}(w)\right), v_e\right)$
       else if $\mathsf{West}(c_1)$ then $\left(\left(\mathsf{W}(w); y_1 + (y_2 - y_1) \cdot \frac{\mathsf{W}(w) - x_1}{x_2 - x_1}\right), v_e\right)$
       else $\left(\left(\mathsf{E}(w); y_1 + (y_2 - y_1) \cdot \frac{\mathsf{E}(w) - x_1}{x_2 - x_1}\right), v_e\right)$
```

Outcode Lemmas

As outcodes play a central role in the Cohen-Sutherland algorithm, they are the starting point for some initial lemmas: If a point is inside the window, it can neither be below, above, left nor right of the window. If it is outside, it cannot be outside at either side. A line is accepted if both points are inside, and it is rejected if both endpoints are outside at a common side.

```
INSIDE_NOTBTLR $\vdash \mathsf{Inside}(c_1) = \neg\mathsf{South}(c_1) \wedge \neg\mathsf{North}(c_1) \wedge \neg\mathsf{West}(c_1) \wedge \neg\mathsf{East}(c_1)$
OUTSIDE_BTLR $\vdash \neg(\mathsf{Inside}(c_1)) = \mathsf{South}(c_1) \vee \mathsf{North}(c_1) \vee \mathsf{West}(c_1) \vee \mathsf{East}(c_1)$
ACCEPT_INSIDE $\vdash \mathsf{Accept}(c_1, c_2) = \mathsf{Inside}(c_1) \wedge \mathsf{Inside}(c_2)$
REJECT_BTLR $\vdash \mathsf{Reject}(c_1, c_2) =$
    $\mathsf{South}(c_1) \wedge \mathsf{South}(c_2) \vee \mathsf{North}(c_1) \wedge \mathsf{North}(c_2) \vee$
    $\mathsf{West}(c_1) \wedge \mathsf{West}(c_2) \vee \mathsf{East}(c_1) \wedge \mathsf{East}(c_2)$
CSA_OUTCODE_TOTAL $\vdash$
    $\mathsf{Inside}(\mathsf{Outcode}(w_1, v_1)) \vee$
    $\mathsf{North}(\mathsf{Outcode}(w_1, v_1)) \vee \mathsf{South}(\mathsf{Outcode}(w_1, v_1)) \vee$
    $\mathsf{West}(\mathsf{Outcode}(w_1, v_1)) \vee \mathsf{East}(\mathsf{Outcode}(w_1, v_1))$
```

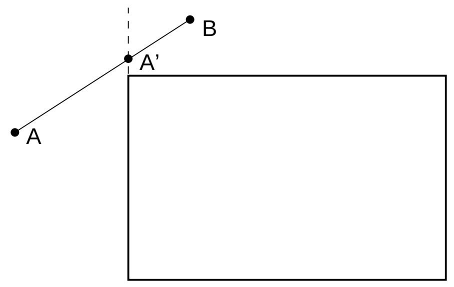

Fig. 25. Counterexample for Proposed Variant

Moreover, if a line segment is not rejected and its first endpoint is outside the window, the other endpoint cannot be outside the window on the same side. On the other hand, if both endpoints are outside the same side of the window, the line segment is rejected.

NOTREJECT_BOTTOM $\vdash \neg\mathsf{Reject}(c_1, c_2) \implies \mathsf{South}(c_1) \implies \neg\mathsf{South}(c_2)$
BOTTOM_REJECT $\vdash \mathsf{South}(c_1) \implies \mathsf{South}(c_2) \implies \mathsf{Reject}(c_1, c_2)$

At this point, the symmetry theorem of Section 4.5 are used to derive analogous theorems for the other three edges of the clipping window. As transformations, the two mirroring of the clipping window are used.

csa_xmirror $\vdash_{\mathrm{def}} \mathsf{csaXmir}(v_1, w_1) =$
$(-\mathsf{X}(v_1) + \mathsf{X}(\mathsf{CornerSW}(w_1)) + \mathsf{X}(\mathsf{CornerNE}(w_1)); \mathsf{Y}(v_1))$
csa_ymirror $\vdash_{\mathrm{def}} \mathsf{csaYmir}(v_1, w_1) =$
$(\mathsf{X}(v_1); -\mathsf{Y}(v_1) + \mathsf{X}(\mathsf{CornerSW}(w_1)) + \mathsf{X}(\mathsf{CornerNE}(w_1)))$

Termination

Since the definition csa_clip is not primitive recursive, its termination is not obvious to the HOL system. On the other side, the number of possible cut operations is obviously limited. At closer inspection of the clipping algorithm, it can be noticed that lines are always cut off in the same order: first at the top, then at the bottom and at the left, and finally at the right. The second remarkable point is the interchange of the endpoints, which is done when the first point of the segment is located inside the window, while the other is outside. The two insights give rise to the following variant:

```
ordered_weight $\vdash_{\mathrm{def}} \mathsf{orderedWeight}(w_1, (v_\mathrm{b}, v_\mathrm{e})) =$
    **let**
        $c_1 = \mathsf{Outcode}(w_1, v_\mathrm{b})$ **and** $c_2 = \mathsf{Outcode}(w_1, v_\mathrm{e})$
    **in**
        $($**if** $\mathsf{North}(c_1)$ **then** $128$ **else** $0) + ($**if** $\mathsf{South}(c_1)$ **then** $64$ **else** $0)+$
        $($**if** $\mathsf{West}(c_1)$ **then** $32$ **else** $0) + ($**if** $\mathsf{East}(c_1)$ **then** $16$ **else** $0)+$
        $($**if** $\mathsf{North}(c_2)$ **then** $8$ **else** $0) + ($**if** $\mathsf{South}(c_2)$ **then** $4$ **else** $0)+$
        $($**if** $\mathsf{West}(c_2)$ **then** $2$ **else** $0) + ($**if** $\mathsf{East}(c_2)$ **then** $1$ **else** $0)$
```

Unfortunately, this function does not have the desired properties. Consider the example shown in Figure 25: According to the definition of orderedWeight, the weight of this situation is $32 + 8 = 40$ After the truncation of the line segment, the TOP bit of the outcode of A is set, and orderedWeight results $128 + 8 = 136$, which is not smaller as required. Hence, the following definition is used instead:

```
line_weight ⊢def lineWeight$(w_1, (v_\mathrm{b}, v_\mathrm{e}))$ =
    let
        $c_1$ = Outcode$(w_1, v_\mathrm{b})$ and $c_2$ = Outcode$(w_1, v_\mathrm{e})$
    in
        ( if $(\neg$Inside$(c_2)$ then 1 else 0)+
        ( if (North$(c_1) \vee$ North$(c_2)$) then 1 else 0)+
        ( if (South$(c_1) \vee$ South$(c_2)$) then 1 else 0)+
        ( if (West$(c_1) \vee$ West$(c_2)$) then 1 else 0)+
        ( if (East$(c_1) \vee$ East$(c_2)$) then 1 else 0)
```

The function lineWeight is used as well-ordering function for the termination proof. This yields two subgoals, which correspond to the two recursive calls of the clipper function. The first one is quite simple: After swapping the endpoints, **if** $\neg$Inside(c_2) **then** 1 **else** 0 changes from 1 to 0, and thus, the variant becomes smaller. For the second subgoal, some more effort is needed: With the predicate CodeMono for an arbitrary outcode predicate C defined above, it is proven that for a given window and a line segment, if the shortened line segment is outside one side, the original line was outside, too. Thus, the weighting function does not become greater.

```
code_mono ⊢def CodeMono$(C, w_1, (v_\mathrm{b}, v_\mathrm{e}))$ =
    let
        $(v'_\mathrm{b}, v'_\mathrm{e})$ = ShortenedLine$(w_1, (v_\mathrm{b}, v_\mathrm{e}))$
    in
        $C($Outcode$(w_1, v'_\mathrm{b})) \vee C($Outcode$(w_1, v'_\mathrm{e}))$ ⟹
        $C($Outcode$(w_1, v_\mathrm{b})) \vee C($Outcode$(w_1, v_\mathrm{e}))$
CSA_CODE_MONO_BOTTOM ⊢
    ¬Accept(Outcode$(w_1, v_\mathrm{b})$, Outcode$(w_1, v_\mathrm{b})$) ⟹
    ¬Reject(Outcode$(w_1, v_\mathrm{b})$, Outcode$(w_1, v_\mathrm{e})$) ⟹
    ¬Inside(Outcode$(w_1, v_\mathrm{b})$) ⟹
    CodeMono(South, $w_1, (v_\mathrm{b}, v_\mathrm{e})$)
```

With the help of theorems of the following form, it is finally shown that the well-ordering function actually becomes smaller. In each recursive call, one part is cut off.

```
BOTTOM_CUT ⊢ ¬North(Outcode$(w_1, v_\mathrm{b})$) ⟹
    South(Outcode$(w_1, v_\mathrm{b})$) ⟹ ¬South(Outcode$(w_1$, ShortenedLineBeg$(w_1, (v_\mathrm{b}, v_\mathrm{e}))$))
```

Correctness

As a first step of the verification, a formal specification of the algorithm is set up: Given a line segment (by its two endpoints v_b and v_e) and a rectangular window (given by its lower left and its upper right corners), return the line segment that is inside the window, where all points on the edge are considered to be inside. To keep the specification concise, the following two predicates are used: CSAInWin(w_1, v) holds if the point v is inside or on the edge of window w_1, and CSAOnSeg$((v_\mathrm{b}, v_\mathrm{e}), v)$ holds if the point v is on the line segment whose endpoints are v_b and v_e (including the endpoints). Note that degenerate inputs and outputs are possible, since the input and the output segments may consist of a single point.

```
csa_in_win $\vdash_{\text{def}}$ $\mathsf{CSAInWin}(w_1, v) =$
    $(\mathsf{inWindow}(w, v) \geq \mathsf{U})$
csa_on_seg $\vdash_{\text{def}}$ $\mathsf{CSAOnSeg}((v_\text{b}, v_\text{e}), v) =$
    if $(v_\text{b} = v_\text{e})$
        then $v_\text{b} = v$
        else $\mathsf{onSegment}(\mathsf{mkLine}(v_\text{b}, v_\text{e}), v) \geq \mathsf{U}$
```

In any case, the result is a value of an option type: it is either None (if the line segment is outside the window) or it represents a pair of points defining the clipped line segment.

```
CSA_CORRECTNESS $\vdash$
    let $r = \mathsf{CSAClip}(w_1, (v_\text{b}, v_\text{e}))$ in
        if $\mathsf{isNone}(r)$ then
            $\neg(\exists v.\mathsf{CSAOnSeg}((v_\text{b}, v_\text{e}), v) \wedge \mathsf{CSAInWin}(w_1, v))$
        else
            $\forall v.\mathsf{CSAOnSeg}(\mathsf{The}(r), v) =$
                $(\mathsf{CSAOnSeg}((v_\text{b}, v_\text{e}), v) \wedge \mathsf{CSAInWin}(w_1, v))$
```

The correctness of the above specification is proven by induction over the recursive calls of the clipping function $\mathsf{CSAClip}(w_1, (v_\text{b}, v_\text{e}))$. There are two base cases: In the first case, the line segment is accepted, since both endpoints are inside the window. It must be proven that every point of the segment is then also in the window (ACCEPT_SEGMENT_INWINDOW).

```
ACCEPT_SEGMENT_INWINDOW $\vdash$
    $\mathsf{Accept}(\mathsf{Outcode}(w_1, v_\text{b}), \mathsf{Outcode}(w_1, v_\text{e})) \implies$
    $\mathsf{CSAOnSeg}((v_\text{b}, v_\text{e}), v) \implies$
    $\mathsf{CSAInWin}(w_1, v)$
```

The Cohen-Sutherland algorithm calculates the intersection point of the clipped line segment and a horizontal or vertical line, respectively.

```
CSA_XLINE_INTERSECT $\vdash \neg(\mathsf{X}(\mathsf{beg}(l_1)) = \mathsf{X}(\mathsf{end}(l_1))) \implies$
    $\mathsf{lineIntersect}(\mathsf{xLine}(r_x), l_1) =$
        let
            $x_1 = \mathsf{X}(\mathsf{beg}(l_1))$ and $y_1 = \mathsf{Y}(\mathsf{beg}(l_1))$ and
            $x_2 = \mathsf{X}(\mathsf{end}(l_1))$ and $y_2 = \mathsf{Y}(\mathsf{end}(l_1))$
        in
            $\left(r_x; y_1 + (y_2 - y_1) \cdot \frac{r_x - x_1}{x_2 - x_1}\right)$
```

The second base case results from the rejection of the line segment in the first recursive call. For the proof, a modified bounding box check CSABoundingBox is used: If the Reject predicate holds, it is proven that the window that is defined by the endpoints of the line segment and the clipping window have no intersection. Thus, the segment does not have an intersection with the clipping window and it is correctly rejected.

`csa_bbox` $\vdash_{\text{def}}$ $\mathsf{CSABoundingBox}(v_1, v_2, v_3, v_4) =$
$\quad(\ \mathsf{vecXint}(v_1, v_3, v_2)\ \ddot{\vee}\ \mathsf{vecXint}(v_1, v_4, v_2)\ \ddot{\vee}$
$\quad\ \ \mathsf{vecXint}(v_3, v_1, v_4)\ \ddot{\vee}\ \mathsf{vecXint}(v_3, v_2, v_4) \geq \mathsf{U}\ )\wedge$
$\quad(\ \mathsf{vecYint}(v_1, v_3, v_2)\ \ddot{\vee}\ \mathsf{vecYint}(v_1, v_4, v_2)\ \ddot{\vee}$
$\quad\ \ \mathsf{vecYint}(v_3, v_1, v_4)\ \ddot{\vee}\ \mathsf{vecYint}(v_3, v_2, v_4) \geq \mathsf{U}\ )$

`REJECT_BBOX_CHECK` $\vdash$
$\quad\mathsf{Reject}(\mathsf{Outcode}(w_1, v_\mathrm{b}), \mathsf{Outcode}(w_1, v_\mathrm{e})) \implies$
$\quad\neg\mathsf{CSABoundingBox}(\mathsf{SW}(w_1), \mathsf{NE}(w_1), v_\mathrm{b}, v_\mathrm{e})$

`CSA_BBOX_CHECK` $\vdash$
$\quad\neg\mathsf{CSABoundingBox}(v_1, v_2, v_3, v_4) \implies$
$\quad(\mathsf{CSAInWin}(\mathsf{mkWin}(v_3, v_4), v) \geq \mathsf{U}) \implies$
$\quad\neg\mathsf{CSAInWin}(v_1, v_2)$

The first recursive call swaps the endpoints. Provided that the induction hypotheses holds, the algorithm is correct, because swapping the endpoints does not change the set of points of the segment (`CSA_ONSEG_SYM`).

`CSA_ONSEG_SYM` $\vdash$
$\quad\mathsf{CSAOnSeg}((v_\mathrm{b}, v_\mathrm{e}), v) = \mathsf{CSAOnSeg}((v_\mathrm{e}, v_\mathrm{b}), v)$

The second recursive call is the hardest part of the proof. The line segment is shortened. Two things must be proven: First, the shortened segment is a subset of the original one (`CSA_CUT_CORRECT`). Second, points on the segment that are inside the window are not cut off (`CSA_CUT_COMPLETE`).

`CSA_CUT_CORRECT` $\vdash$
$\quad\neg\mathsf{Accept}(\mathsf{Outcode}(w_1, v_\mathrm{b}), \mathsf{Outcode}(w_1, v_\mathrm{e})) \implies$
$\quad\neg\mathsf{Reject}(\mathsf{Outcode}(w_1, v_\mathrm{b}), \mathsf{Outcode}(w_1, v_\mathrm{e})) \implies$
$\quad\neg\mathsf{Inside}(\mathsf{Outcode}(w_1, v_\mathrm{b})) \implies$
$\quad\mathsf{CSAInWin}(w_1, v) \implies$
$\quad\mathsf{CSAOnSeg}(\mathsf{ShortenedLine}(w_1, (v_\mathrm{b}, v_\mathrm{e})), v) \implies$
$\quad\mathsf{CSAOnSeg}((v_\mathrm{b}, v_\mathrm{e}), v)$

`CSA_CUT_COMPLETE` $\vdash$
$\quad\neg\mathsf{Accept}(\mathsf{Outcode}(w_1, v_\mathrm{b}), \mathsf{Outcode}(w_1, v_\mathrm{e})) \implies$
$\quad\neg\mathsf{Reject}(\mathsf{Outcode}(w_1, v_\mathrm{b}), \mathsf{Outcode}(w_1, v_\mathrm{e})) \implies$
$\quad\neg\mathsf{Inside}(\mathsf{Outcode}(w_1, v_\mathrm{b})) \implies$
$\quad\mathsf{CSAInWin}(w_1, v) \implies$
$\quad\mathsf{CSAOnSeg}((v_\mathrm{b}, v_\mathrm{e}), v) \implies$
$\quad\mathsf{CSAOnSeg}(\mathsf{ShortenedLine}(w_1, (v_\mathrm{b}, v_\mathrm{e})), v)$

`LINE_SPLIT` $\vdash$
$\quad(\mathsf{onSegment}(l_1, v_m) = \mathsf{T}) \implies$
$\quad\neg(v_m = \mathsf{end}(l_1)) \implies$
$\quad((\mathsf{onSegment}(l_,)\ 1v \geq \mathsf{U}) =$
$\quad\quad(v = \mathsf{beg}(l_1))\vee$
$\quad\quad(\mathsf{onSegment}(\mathsf{mkLine}(\mathsf{beg}(l_1), v_m), v) = \mathsf{T})\vee$
$\quad\quad(\mathsf{onSegment}(\mathsf{mkLine}(v_m, \mathsf{end}(l_1)), v) \geq \mathsf{U}))$

The key to prove the remaining part is to show that the line segment is split into two partitions (`LINE_SPLIT`). This concludes the correctness proof that holds for all cases (including that endpoints are on the edges of the window, or that both endpoints are the same). Hence, it is proven that the algorithm terminates and that it returns the specified result.

4.6 Dependent Objects

Some computational geometry algorithms only classify a given set of input objects, while some other create new objects. These output objects are usually constructed from other objects. Properties for these derived objects can be determined by only considering the previously existing objects. This is extremely useful for verification procedures: Considering the construction of an object increases the abstraction level in proofs, since properties do not have to be proven from scratch. Instead, they can be derived from the original objects and the construction. Thus, readable proofs for complicated examples can be derived and general proof strategies can be applied to find a solution in the proof space. However, in order to benefit from the constructions, the proof goals must follow a special form: First, the property to be shown must be given in terms of some primitives that can be applied in this approach. Second, the geometric situation must be defined with some free points and constructions on top of these points. Under these preconditions, proofs can follow the constructions in reverse order and eliminate occurrences of the constructed points in the goal, until it only contains general points.

4.6.1 Proofs with Dependent Objects

There are existing proof procedures that exploit this construction principle. The most important among them is certainly the one by Chou, Gao and Zhang [29, 139], which has been already implemented within the general theorem prover Coq by Narboux [107]. The verification approach presented in this section is inspired by this work. However, in contrast to it, constructions are not seen as a basis of the formalisation of geometry, which lead to the freedom to consider other constructions and predicates. Moreover, three-valued definitions and lemmas are used in compliance with the verification framework.

The resulting tactic is not implemented as a fully automated decision procedure. However, with the general high-level tactics of the HOL system, which are used to identify the appropriate elimination lemmas, most goals can be automatically reduced to an equality between two rational expressions, which was described in Section 2.2.

Polygon processing generally deals with lines and points. In the following, their construction from existing ones is considered. In particular, the following operations are used:

$$\texttt{vec\_on\_line} \vdash_{\text{def}} \mathsf{vecAt}\,(l_1, r_1) = \mathsf{beg}(l_1) + r_1 \cdot \mathsf{dirvec}(l_1)$$

$$\texttt{line\_intersect} \vdash_{\text{def}}$$

$$\mathsf{lineIntersect}(l_1, l_2) = \mathsf{vecAt}\left(l_2, \frac{(\mathsf{beg}(l_1) - \mathsf{beg}(l_2)) \times \mathsf{dirvec}(l_1)}{\mathsf{dirvec}(l_2) \times \mathsf{dirvec}(l_1)}\right)$$

$$\texttt{parallelLine} \vdash_{\text{def}} \mathsf{parallelLine}(l_1, v_1) = \mathsf{mkLine}(v_1, v_1 + \mathsf{dirvec}(l_1))$$

The term $\mathsf{vecAt}\,(l_1, \lambda)$ returns a point on the line l_1 at a position that is specified by λ. The intersection of l_1 and l_2 is returned by $\mathsf{lineIntersect}(l_1, l_2)$ if it exists. The application of $\mathsf{parallelLine}(l_1, v_1)$ generates a line parallel to l_1 through the point v_1.

A very important primitive that can be used with constructions are the area primitives triArea and quadArea. They have been used to define the lturn predicate. Other predicates of Section 4 can be also formulated with them as the following theorems assert:

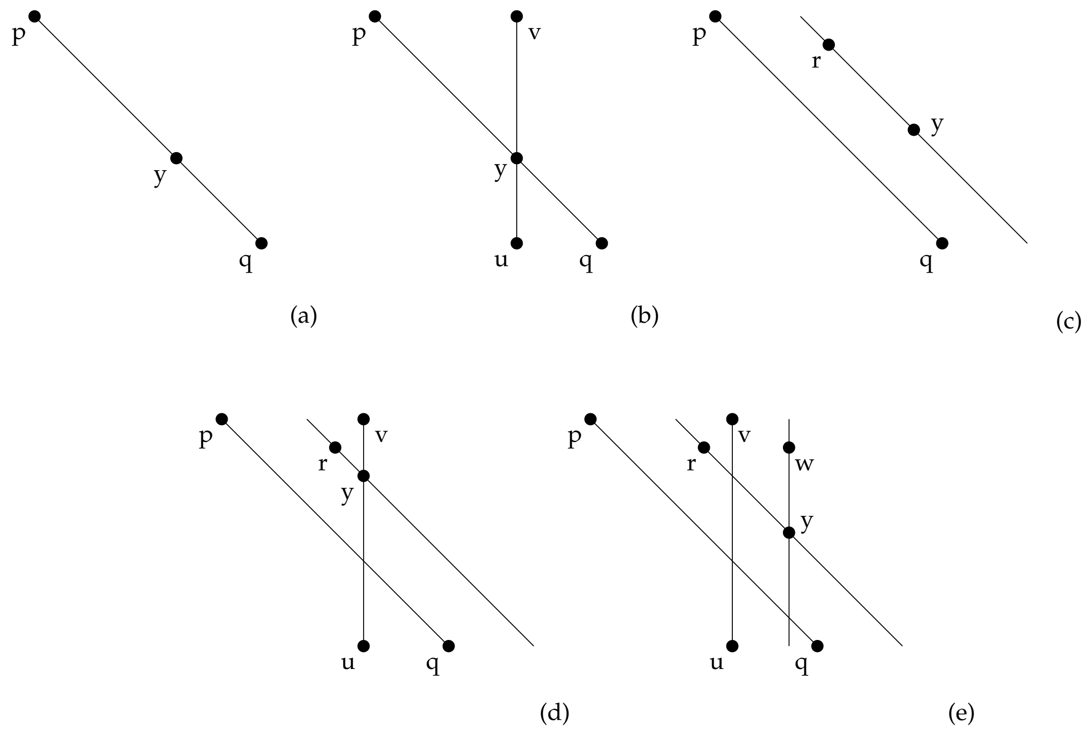

Fig. 26. Constructions of Dependent Points

$$\begin{aligned}
&\texttt{AREA\_COLINEAR} \vdash \\
&\quad \mathsf{collinear}(v_1, v_2, v_3) = (\mathsf{triArea}(v_1, v_2, v_3) = 0) \\
&\texttt{AREA\_PARALLEL} \vdash \\
&\quad \mathsf{parallel}(l_1, l_2) = (\mathsf{triArea}(\mathsf{beg}(l_1), \mathsf{end}(l_1), \mathsf{beg}(l_2)) = \mathsf{triArea}(\mathsf{beg}(l_1), \mathsf{end}(l_1), \mathsf{end}(l_2))) \\
&\texttt{AREA\_MIDPOINT} \vdash \mathsf{onLine}(l_1, v_2) \implies \\
&\quad \mathsf{isMidpoint}(l_1, v_1) = (\mathsf{triArea}(\mathsf{beg}(l_1), v_1, v_2) = \mathsf{triArea}(v_1, \mathsf{end}(l_1), v_2))
\end{aligned}$$

Another important primitive is the ratio between the length of two line segments. As there is no length function, a relative distance is defined, which equals to the intended meaning for parallel segments.

$$\texttt{relDist} \vdash_{\mathrm{def}} \mathsf{relDistance}(l, a, b) = \frac{(b-a) \circ \mathsf{dirvec}(l)}{\mathsf{dirvec}(l) \circ \mathsf{dirvec}(l)}$$

In principle, the sets of primitives and constructions can be arbitrarily defined. For a universal proof method, both should be kept small, since for each element of the Cartesian product of both sets, a lemma must be proven, which reformulates the primitive by substituting the derived object by its construction. This lemma is used to eliminate the constructed point

The proof generally follows the following scheme: To eliminate a point from the goal, all primitives containing it are rewritten by an elimination lemma. Finally, an expression using only free points, which can be arbitrarily chosen, is derived. This expression must be proven with a decision procedure or the help of lemmas, e. g. the ones presented in Section 4 for the orientation and position predicates.

Figure 26 shows the constructions which are considered in the following: (a) a point on a line at position λ, (b) the intersection of two lines, (c) a point on a parallel line, (e) the intersection of a parallel line and an input line and (e) the intersection of parallel lines of input lines.

A triangle primitive triArea that contains a dependent point y in its arguments can be substituted by area primitives with only free points as the following elimination state. In all these lemmas, y is the third argument, which can be generally assumed, since the arguments can be rotated by the LTURN_SYM property (see Section 4.4).

TRI_ELIMINATION_A $\vdash$

$$\begin{array}{l}\textbf{let } p = \mathsf{beg}(l_1) \textbf{ and } q = \mathsf{end}(l_1) \textbf{ in}\\ y = \mathsf{vecAt}\,(l_1, \lambda) \implies\\ \quad \mathsf{triArea}(a,b,y) = (1-\lambda)\cdot \mathsf{triArea}(a,b,p) + \lambda \cdot \mathsf{triArea}(a,b,q)\end{array}$$

TRI_ELIMINATION_B $\vdash$

$$\begin{array}{l}\textbf{let } p = \mathsf{beg}(l_1) \textbf{ and } q = \mathsf{end}(l_1) \textbf{ and } u = \mathsf{beg}(l_2) \textbf{ and } v = \mathsf{end}(l_2) \textbf{ in}\\ y = \mathsf{lineIntersect}(l_1, l_2) \implies\\ \quad \mathsf{triArea}(a,b,y) = \dfrac{\mathsf{triArea}(p,u,v)\cdot \mathsf{triArea}(a,b,q) + \mathsf{triArea}(q,v,u)\cdot \mathsf{triArea}(a,b,p)}{\mathsf{quadArea}(p,u,q,v)}\end{array}$$

TRI_ELIMINATION_C $\vdash$

$$\begin{array}{l}\textbf{let } p = \mathsf{beg}(l_1) \textbf{ and } q = \mathsf{end}(l_1) \textbf{ in}\\ l_2 = \mathsf{parallelLine}(l_1, r) \implies\\ y = \mathsf{vecAt}\,(l_2, \lambda) \implies\\ \quad \mathsf{triArea}(a,b,y) = \mathsf{triArea}(a,b,r) + \lambda \cdot \mathsf{quadArea}(a,p,b,q)\end{array}$$

TRI_ELIMINATION_D $\vdash$

$$\begin{array}{l}\textbf{let } p = \mathsf{beg}(l_1) \textbf{ and } q = \mathsf{end}(l_1) \textbf{ and } u = \mathsf{beg}(l_2) \textbf{ and } v = \mathsf{end}(l_2) \textbf{ in}\\ \neg\mathsf{parallel}(l_1, l_2) \implies\\ l_3 = \mathsf{parallelLine}(l_1, r) \implies\\ y = \mathsf{lineIntersect}(l_2, l_3) \implies\\ \quad \mathsf{triArea}(a,b,y) = \dfrac{\mathsf{quadArea}(p,u,q,r)\cdot \mathsf{triArea}(a,b,u) - \mathsf{quadArea}(p,v,q,r)\cdot \mathsf{triArea}(a,b,u)}{\mathsf{quadArea}(p,u,q,v)}\end{array}$$

TRI_ELIMINATION_E $\vdash$

$$\begin{array}{l}\textbf{let } p = \mathsf{beg}(l_1) \textbf{ and } q = \mathsf{end}(l_1) \textbf{ and } u = \mathsf{beg}(l_2) \textbf{ and } v = \mathsf{end}(l_2) \textbf{ in}\\ \neg\mathsf{parallel}(l_1, l_2) \implies\\ l_3 = \mathsf{parallelLine}(l_1, p_3) \implies\\ l_4 = \mathsf{parallelLine}(l_1, p_4) \implies\\ y = \mathsf{lineIntersect}(l_3, l_4) \implies\\ \quad \mathsf{triArea}(a,b,y) = \dfrac{\mathsf{quadArea}(p,w,q,r)\cdot \mathsf{quadArea}(a,u,b,v)}{\mathsf{quadArea}(p,u,b,v)} + \mathsf{triArea}(a,b,w)\end{array}$$

Similar elimination lemmas can be given for the relative distance. The relDistance primitive for a constructed and a free point can be replaced by triArea primitives according to the following lemmas.

DIST_ELIMINATION_A $\vdash$

$$\begin{array}{l}
\textbf{let } p = \mathsf{beg}(l_1) \textbf{ and } q = \mathsf{end}(l_1) \textbf{ in}\\
\neg\mathsf{onLine}(l_1, a) \implies\\
y = \mathsf{vecAt}\,(l_1, \lambda) \implies\\
\quad \mathsf{relDistance}(l_0, a, y) = \dfrac{\mathsf{triArea}(a, p, q)}{\mathsf{quadArea}(c, p, d, q)}
\end{array}$$

DIST_ELIMINATION_B $\vdash$

$$\begin{array}{l}
\textbf{let } p = \mathsf{beg}(l_1) \textbf{ and } q = \mathsf{end}(l_1) \textbf{ and } u = \mathsf{beg}(l_2) \textbf{ and } v = \mathsf{end}(l_2) \textbf{ in}\\
\neg\mathsf{onLine}(l_2, a) \implies\\
y = \mathsf{lineIntersect}(l_1, l_2) \implies\\
\quad \mathsf{relDistance}(l_0, a, y) = \dfrac{\mathsf{triArea}(a, u, v)}{\mathsf{quadArea}(c, u, d, v)}
\end{array}$$

DIST_ELIMINATION_C $\vdash$

$$\begin{array}{l}
\textbf{let } p = \mathsf{beg}(l_1) \textbf{ and } q = \mathsf{end}(l_1) \textbf{ in}\\
\neg\mathsf{onLine}(l_2, a) \implies\\
l_2 = \mathsf{parallelLine}(l_1, r) \implies\\
y = \mathsf{vecAt}\,(l_2, \lambda) \implies\\
\quad \mathsf{relDistance}(l_0, a, y) = \dfrac{\mathsf{quadArea}(a, p, r, q)}{\mathsf{quadArea}(c, p, d, q)}
\end{array}$$

DIST_ELIMINATION_D $\vdash$

$$\begin{array}{l}
\textbf{let } p = \mathsf{beg}(l_1) \textbf{ and } q = \mathsf{end}(l_1) \textbf{ and } u = \mathsf{beg}(l_2) \textbf{ and } v = \mathsf{end}(l_2) \textbf{ in}\\
\neg\mathsf{onLine}(l_2, a) \implies\\
\neg\mathsf{parallel}(l_1, l_2) \implies\\
l_3 = \mathsf{parallelLine}(l_1, r) \implies\\
y = \mathsf{lineIntersect}(l_2, l_3) \implies\\
\quad \mathsf{relDistance}(l_0, a, y) = \dfrac{\mathsf{triArea}(a, u, v)}{\mathsf{quadArea}(c, u, d, v)}
\end{array}$$

DIST_ELIMINATION_E $\vdash$

$$\begin{array}{l}
\textbf{let } p = \mathsf{beg}(l_1) \textbf{ and } q = \mathsf{end}(l_1) \textbf{ and } u = \mathsf{beg}(l_2) \textbf{ and } v = \mathsf{end}(l_2) \textbf{ in}\\
\neg\mathsf{onLine}(l_3, a) \implies\\
\neg\mathsf{parallel}(l_1, l_2) \implies\\
l_3 = \mathsf{parallelLine}(l_1, p_3) \implies\\
l_4 = \mathsf{parallelLine}(l_1, p_4) \implies\\
y = \mathsf{lineIntersect}(l_3, l_4) \implies\\
\quad \mathsf{relDistance}(l_0, a, y) = \dfrac{\mathsf{quadArea}(a, p, r, q)}{\mathsf{quadArea}(c, p, d, q)}
\end{array}$$

In the algebraic geometry theorem proving approach of Section 3.4 nondegenerate conditions play an important role. This is also the case for the above constructive approach here. These conditions are given in the assumptions, either explicitly as by $\neg\mathsf{parallel}(l_1, l_2)$ or implicitly by the properties of each line ($\mathsf{beg}(l_1) \neq \mathsf{end}(l_1)$).

4.6.2 Diagonals of a Parallelogram

The example of Section 3.4 is considered here once again. However, a higher level of abstraction is taken now. With help of the above constructions, the diagonal theorem can be proven without the analytic geometry foundations.

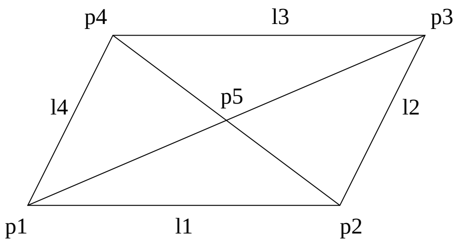

Fig. 27. Parallelogram and its Diagonals

It should be proven that the intersections of the diagonals intersect each other in their middle, i. e. the ratio between the lengths of the segments $\overline{p_1 p_5}$ and $\overline{p_1 p_3}$ is $\frac{1}{2}$. The first step to prove this fact is to construct the dependent objects from independent ones with the operations defined in the previous section.

In the parallelogram, three points are free p_1, p_2 and p_3. The fourth one p_4 is dependent and is constructed as the intersection of two parallels of the segments induced by p_1, p_2 and p_3. The point p_5 in the middle is the intersection of the diagonals. Thus, the goal is as follows:

$$
\begin{array}{l}
\texttt{PARALLELOGRAM\_DIAGONALS} \vdash^{?} \\
\quad (\mathsf{beg}(l_1) = p_1) \implies (\mathsf{end}(l_1) = p_2) \implies \\
\quad (\mathsf{beg}(l_2) = p_2) \implies (\mathsf{end}(l_2) = p_3) \implies \\
\quad \neg\mathsf{parallel}(l_1, l_2)) \implies \\
\quad (l_3 = \mathsf{parallelLine}(l_1, p_3)) \implies \\
\quad (l_4 = \mathsf{parallelLine}(l_2, p_1)) \implies \\
\quad (p_4 = \mathsf{lineIntersect}(l_3, l_4)) \implies \\
\quad (d_1 = \mathsf{mkLine}(p_1, p_3)) \implies \\
\quad (d_2 = \mathsf{mkLine}(p_2, p_4)) \implies \\
\quad (p_5 = \mathsf{lineIntersect}(d_1, d_2) \implies \\
\quad \mathsf{relDistance}(d_1, p_1, p_5) = \frac{1}{2}
\end{array}
$$

This goal is solved by applying `DIST_ELIMINATION_B` to eliminate p_5 from the goal. Subsequently, p_4 can be removed by `TRI_ELMINATION_E`.

4.7 Implementation of Three-Valued Primitives

For the implementation, the definitions of the previous sections are translated one-to-one. To use the same techniques for the specification and the implementation has various obvious advantages: First, the specification and the actual implementation are as close as possible, since all primitives are implemented in software in the same way as they were defined in the HOL theories. Second, algorithms can also benefit from the same advantages of the three-valued primitives: They are more compact, since several cases of the two-valued formalisation can be merged in the three-valued setting.

4.7.1 Three-valued Predicates

Before three-valued primitives can be implemented, the three-valued logic must be integrated in the polygon library. The main design decision in this context is the choice of the underlying data type.

$x \in \mathbb{B}^2$	$\epsilon(x) \in \mathbb{T}$
$(0,1)$	F
$(0,0)$	U
$(1,0)$	T

Fig. 28. Dual-Rail Encoding

Dual-rail encoding, which was originally presented [12], is a wide-spread possibility for this purpose. It uses two Booleans to encode a ternary value so that binary decision diagrams and other data structures and algorithms can be used for ternary values as well. [13, 125]. The mapping of the three truth values is done according to the table shown in Figure 28.

Using dual-rail encoding, a ternary function $f : \mathbb{T}^n \to \mathbb{T}$ is represented by two Boolean functions $(g(x_1, x_2), h(x_1, x_2))$, as

- $\neg_2(x_1, x_2) = (x_2, x_1)$
- $(x_1, x_2) \wedge_2 (y_1, y_2) = (x_1 \wedge y_1, x_2 \vee y_2)$
- $(x_1, x_2) \vee_2 (y_1, y_2) = (x_1 \vee y_1, x_2 \wedge y_2)$

Thus, all operations are reduced to two-valued ones. However, this encoding is not optimal for the polygon-processing library. A better representation of the three truth values are signed numbers: F is mapped to -1, U to 0 and T to 1, where the smallest supported type is chosen as a base (which is usually `char` in C).

```
typedef signed char log3;
#define FF -1
#define UU 0
#define TT 1
```

Negation $\ddot{\neg}$ is implemented by reversing the sign, conjunction $\ddot{\wedge}$ is the minimum of two arguments, disjunction $\ddot{\vee}$ the maximum, and equivalence $\ddot{\leftrightarrow}$ can be computed by multiplying the arguments. Implication $\ddot{\rightarrow}$ and antivalence $\ddot{\oplus}$ are reduced with the help of negation to disjunction and equivalence. All other operators like $\ddot{*}$ or $\twoheadrightarrow$ are implemented by their respective truth tables as follows:

```
log3 awa3_table[] = {{FF,FF,FF},{FF,UU,TT},{FF,TT,TT}};
log3 awa3( log3 a, log3 b ) { return awa3_table[a+1][b+1] }
```

All topological primitives can be computed straightforwardly. For instance, the existence of an intersection point of two line segments is implemented as follows:

```
log3 seg_do_intersect( line l1, line l2 )
{
  return and3(
    equ3(
      lturn( l1.beg,l1.end,l2.beg ),
      rturn( l1.beg,l1.end,l2.end )
    ),(
      lturn( l2.beg,l2.end,l1.beg ),
      rturn( l2.beg,l2.end,l1.end )
   ))
}
```

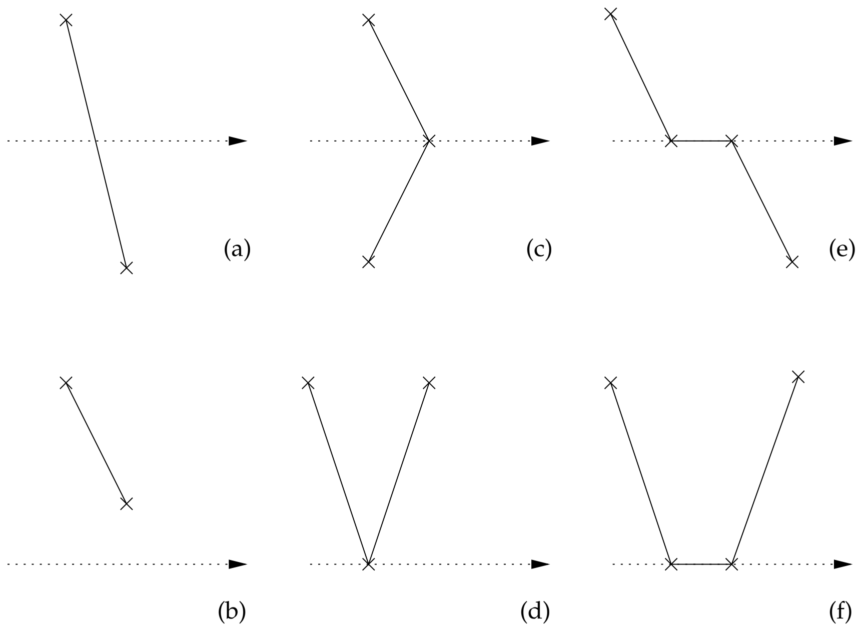

Fig. 29. Winding Number Algorithm

If both segments truly intersect, T is returned. If they only touch or have a common line segment, U is returned, since this is a degenerated case. If there are no common points, F is returned.

So far, the encoding of the truth values has not been exploited. The reason for this encoding becomes clear when sequences of objects are processed. Then, the results of individual primitives can be simply added. For instance, consider the *winding number* algorithm: It calculates the winding number of a point v relative to a polygon P, i.e. how often the directed sequence of connected line segments of P turns around v (see Section 6.1). To this end, it counts the intersections of an arbitrary ray starting in v with edges of the polygon P. If the intersected edge runs from the bottom to the top, it is counted positively - in the opposite direction negatively. Figure 29 shows the possible cases, where the ray is drawn with a dotted line and some edges of polygon P are drawn with straight lines: (a) and (b) show simple cases without problems, while (c) to (f) show degenerate cases, i.e. a vertex or an edge of the polygon is on the ray. Depending on the position of the adjoining edges of the polygon, the situation must be counted as an intersection or not. In the presented examples, cases (c) and (e) are counted as an intersection, whereas cases (d) and (f) are not counted as intersections.

A classic implementation of this algorithm that directly deals with degenerate cases cannot avoid the case distinctions shown in Figure 29. The position of the previous and following points must be taken into account, which leads to even more subcases. With the help of a three-valued intersection, the algorithm can be formulated much simpler, since the addition and three-valued primitives hide all degeneracies.

Provided that $e[i]$ denotes the i-th of n edges of a polygon and `xRay`(v) is the ray from v to the right, the following C fragment calculates the winding number w of a point v:

```
w = 0;
for( i = 0 ; i < n ; i++ )
    if( ray_do_intersect( xRay(v),e[i] ) >= UU ) {
        w += below( v, e[i].beg );
        w += above( v, e[i].end );
    }
    w = w / 2;
```

4.7.2 Geometric Objects

The following header fragment gives an impression about the implementation of geometric objects. It describes the most important functions defined for lines and segments. They are both represented by the C type `epp_line`. As for every object of the polygon-processing library, there are constructors and destructors to allocate memory and initialise the components or free the memory afterwards. Moreover, lines can be copied — either to a given or a newly created target. The next group of functions implements the predicates and the constructions that are defined for lines and segments in this chapter.

```
struct epp_line_t { epp_vec *beg; epp_vec *end; };
typedef struct epp_line_t epp_line;

void epp_line_init( epp_line *l1 );
epp_line* epp_line_create();
void epp_line_cleanup( epp_line *l1 );
void epp_line_destroy( epp_line *l1 );

void epp_line_set( epp_line *l0, const epp_line *l1 );
epp_line* void epp_line_copy( const epp_line *l1 );

log3 epp_line_lturn( const epp_vec *v1, const epp_vec *v2, const epp_vec *v3 );
log3 epp_line_lofline( const epp_line *l1, const epp_vec *v1 );
unsigned char epp_line_parallel( const epp_line *l1, const epp_line *l2 );
unsigned char epp_line_equiv( const epp_line *l1, const epp_line *l2 );
log3 epp_line_on_line( const epp_line *l1, const epp_vec *v1 );
log3 epp_line_on_seg( const epp_line *l1, const epp_vec *v1 );

void epp_line_veconline( epp_vec *v0, const epp_line *l1, const epp_rat *r1 );
log3 epp_line_intersect( epp_vec *v0, const epp_line *l1, const epp_line *l2 );
log3 epp_line_seg_intersect( epp_line *l0, const epp_line *l1, const epp_line *l2 );
```

4.7.3 Arithmetics

As discussed in Section 3.1, imprecisions of the underlying arithmetic must be considered when implementing geometric primitives.

Extending the Precision of Computations

Various researchers have already addressed the precision problem, and some general remedies have been proposed: For geometric predicates, a common approach is to extend the precision of the computations in up to the point where a precise answer can be given. Floating point filters [51, 52, 53] or adaptive-precision arithmetic [123] are two ways to implement this principle. They are based on the fact that it is frequently possible to determine the sign of an expression without computing its precise value. By using some kind of interval arithmetic, the result can be approximated, and the intervals are iteratively refined. Compared to arbitrary-precision arithmetic, the runtime for the computation of a geometric predicate is only moderately increased.

However, there are some drawbacks: First, formal verification within a theorem prover is not practical for this work, since the additional effort to implement all the required mathematics for them is substantial. Hence, one must rely on the paper and pencil proofs by the original authors. Second, they do not give an answer to constructed objects, similar to symbolic perturbations. For instance, intersection points of two lines that are given by two points with precision n each, require about $4n$ precision. Without falling back to arbitrary precision primitives, there is only one solution: to restrict the precision of input objects.

Restricting the Precision of Inputs

If geometric objects are given in terms of fixed-precision rationals, their position is based on finitely many points in the plane. These points form a fixed grid that represents all representable positions. New objects, which have been constructed in the course of an algorithm, must be aligned to this grid.

On this level, however, objects cannot be aligned to a grid. Since a modification of positions should not change the topological situation, the geometric context must be respected. Thus, this problem is deferred again to an upper layer of the system that is aware of the whole situation. It can decide how input objects can be aligned without compromising the topological structure of a geometric situation. In this layer, arbitrary-precision rational numbers are assumed and the problem will be reconsidered in Chapter 6.

Chapter 5

Plane Sweep Algorithms

A lot of computational geometry algorithms follow some common execution schemes. One of these schemes is the plane sweep paradigm, which computes the results while sweeping a line from left to the right over the plane. [1] While moving the line, the algorithm keeps track of all changes that result from objects that currently intersect the sweep line. Therefore, it belongs to the class of incremental algorithms, which compute their results in discrete steps that are induced by their inputs. These algorithms tend to be simpler and more efficient, since each of these steps only affects the local context. Decisions are made with a limited scope, and modifications only affect this scope.

Section 5.1 presents the general paradigm. After presenting a verification framework for plane sweep algorithms in Section 5.2, it is applied to the network overlay in Sections 5.3 and 5.4. Finally, Section 5.5 presents some implementation details.

5.1 General Paradigm

Generally, plane sweep algorithms use two data structures. First, they maintain a *status structure*, which reflects the situation at the current position of the sweep line. Usually, this is an ordered sequence of the objects that intersect the sweep line. Initially, when the sweep line is left of all objects that are processed, it is empty. Every time, the sweep line detects a change, the status structure is updated.

These changes are detected by the second basic data structure, the *event queue*. Initially, the queue contains some information about the input objects. Each of its elements $q = (v, e)$ consists of an event e and its geometric position $v = \mathsf{position}(e)$, which gives the element its place in the queue: The elements are sorted in the order that the sweep line touches them when sweeping the plane. Events should be defined in a way that the geometric situation with respect to the given problem only changes at these event points so that the algorithm can update the status structure.

Degenerate cases for the plane sweep algorithm are events that have the same x-coordinate. This is usually solved, by arranging the event in lexical ordering, which arranges events like in a dictionary, i. e. events are first sorted by their x-coordinates, and ties are resolved by comparing the y-coordinates:

$$\texttt{vec\_lex} \vdash_{\text{def}} v_1 \prec v_2 = \textbf{if } (\mathsf{X}(v_1) \neq \mathsf{X}(v_2)) \textbf{ then } (\mathsf{X}(v_1) \prec \mathsf{X}(v_2)) \textbf{ else } (\mathsf{Y}(v_1) \prec \mathsf{Y}(v_2))$$

[1] The plane sweep algorithm can be generalised to higher dimensions by e. g. sweeping a plane in some direction through the space.

```
function EventQueueEmpty($\mathcal{Q}$)
    return $\mathcal{Q} = \langle\rangle$;

function EventQueuePop($\mathcal{Q}$)
    $\mathcal{Q} = \mathsf{TL}(\mathcal{Q})$;
    return $\mathsf{HD}(\mathcal{Q})$;

function EventQueuePush($\mathcal{Q}, (v_1, e_1)$)
    switch
    case $\mathcal{Q} = \langle\rangle$ :
        return $\langle(v, \{e_1\})\rangle$;
    case $\mathcal{Q} = (v_0, E_0) :: \mathcal{Q}$ :
        switch
        case $v_0 \prec v_1$ :
            return $(v_0, E_0) :: \mathsf{EventQueueInsert}(\mathcal{Q}, (v_1, e_1))$;
        case $v_0 = v_1$ :
            return $(v, \mathcal{E}_0 \cup \{e_1\}) :: \mathcal{Q}$;
        case $v_0 \succ v_1$ :
            return $(v, \{e_1\}) :: (v, \{e_1\}) :: \mathcal{Q}$;
```

Fig. 30. Operations on Event Queues

With this ordering, vertical line segments can be seen as perturbed, so that their lower endpoint are a little bit further left than their corresponding upper endpoints. In contrast to other publications, this is not considered as a perturbation in the sense of Section 4.2.2, but as a well-defined order. However, this ordering does not solve all problems. Two event points can still have the same position. Therefore, a three-valued approach is used again, and the event queue is extended: Instead of events, it contains sets of events $q = (v, E)$, where two events are in the same set if and only if their positions are equal:

$$\mathcal{Q} = \langle q_0, \ldots, q_n\rangle \textbf{ where } q_i = (v_i, \{e_i^0, \ldots, e_i^m\})$$
$$\textit{with } \mathsf{position}(e_i^j) = \mathsf{position}(e_i^k) \textit{ for } 0 \le i \le n \textit{ and } 0 \le j, k \le m$$

The event queue offers three operations (see Figure 30): First, there is EventQueueEmpty, which checks whether the event queue is empty. Second, EventQueuePop can be used to obtain the top element of the queue and remove it from the queue. This set contains the leftmost events right from the current position of the sweep line. If the queue is empty, this function fails. Third, new events that are determined in the course of the computation, can be inserted with respect to the event queue order by EventQueuePush. It can be seen that this operation distinguishes three cases: A position v_0 can be either smaller, greater or equal to another position v_1. If other events with the same position exist in the event queue, the new event is inserted in their set. Otherwise, a new element in the event queue is inserted.

The status structure, on the other side, has also to support insertion and deletion operations, but the actual set of operations depends on the used algorithm. However, each status structure is based on a invariant order, which must be kept when performing any changes on the structure.

Figure 31 shows the general structure of a plane sweep algorithm. First, the event queue Q is filled with input elements $I_1, \ldots., I_n$, and the status structure S is initialised. In each iteration

```
function PlaneSweep($I_1, \ldots, I_n$)
    $\mathcal{Q}$ = InitialiseEventQueue($I_1, \ldots, I_n$);
    $S = \langle\rangle$;
    while ($\mathcal{Q} \neq \langle\rangle$)
        $E$ = HD ($\mathcal{Q}$) ;
        $\mathcal{Q}$ = TL ($\mathcal{Q}$) ;
        HandleEvent($E, S$);
```

Fig. 31. General Structure of Plane Sweep Algorithms

of the main loop, the first element of the event queue is removed. The handling of the event changes the status structure and may generate new events, which are inserted at the appropriate position of the event queue. Note that the skeleton shown in Figure 31 omits the computation of the result of the algorithm, which is given by actual algorithm implementing the plane sweep approach. These computations are triggered by the events and thus, they are implemented in the HandleEvent function.

The complexity of the plane sweep algorithms tends to be better than equivalent naive approaches, provided that the operations on the main data structures are implemented efficiently. This is usually done with the help of balanced trees, which offer all the needed operations with complexity $O(\log n)$. If all the events can be handled in constant time, an optimal run-time complexity of $O(n \log n)$ is obtained.

5.2 Reasoning About Plane Sweep Algorithms

5.2.1 Formalisation

The previous section presented the general concepts of the plane sweep approach. They cannot only be reused as a pattern for the algorithm design or as functions in a computational geometry library, but also basic definitions can be integrated in the verification framework. Related proof tools like theorems, tactics and conversions to automate parts of the reasoning about these objects are then provided so that formalisations of new algorithms can use them.

The event queue is the first structure that is integrated. It always consists of a list of lexically ordered points and a number of events that are associated with these points. The event queue is derived from the type $(\mathbb{R}^2 \times \alpha\,\mathsf{list})\,\mathsf{list}$, where α is the type of events [2]. Not all lists are valid event queues, so the new type is restricted to sorted event queues.

`equeue_compare` $\vdash_{\text{def}}$

$$\mathsf{equeueCompare}(e_1, e_2) = \textbf{let } (v_1, E_1) = e_1 \textbf{ and } (v_2, E_2) = e_2 \textbf{ in } v_1 \prec v_2$$

`equeue_sorted` $\vdash_{\text{def}}$

$$(\mathsf{equeueSorted}(\langle\rangle) = \mathsf{T}) \wedge$$
$$(\mathsf{equeueSorted}(\langle e\rangle) = \mathsf{T}) \wedge$$
$$(\mathsf{equeueSorted}(e_1 :: e_2 :: Q) = (\mathsf{equeueCompare}(e_1, e_2) = \mathsf{T}) \wedge \mathsf{equeueSorted}(e_2 :: Q))$$

[2] Although the order of the events of single event point is not relevant, they are stored in a list instead of a set, since lists are usually more friendly for theorem proving.

Inserting and removing elements from the event queue are frequent operations for many algorithms. The definitions follow the description given in the previous section (see Figure 30). Basic list operations are transferred to event queues as shown in `equeue_empty`, `equeue_make` and `equeue_cons`. (absQ and repQ denote the abstraction and the representation function for event queues.)

`equeue_empty` $\vdash_{\mathrm{def}} \langle\rangle = \mathsf{absQ}(\langle\rangle)$
`equeue_make` $\vdash_{\mathrm{def}} \langle E\rangle = \mathsf{absQ}(\langle E\rangle)$
`equeue_cons` $\vdash_{\mathrm{def}} E :: Q = \mathsf{absQ}(E :: \mathsf{repQ}(Q))$
`equeue_empty` $\vdash_{\mathrm{def}} (\mathsf{equeueEmpty}(\langle\rangle) = \mathsf{T}) \wedge$
$(\mathsf{equeueEmpty}(e :: Q) = \mathsf{F})$
`equeue_push` $\vdash_{\mathrm{def}} \mathsf{equeuePush}((v_1, e_1), \langle\rangle) = \langle(v_1, \langle e_1\rangle)\rangle$

```
$\mathsf{equeuePush}((v_1, e_1), (v_0, E_0) :: Q) =$
  **switch**
    **case** $v_0 \prec v_1 = \mathsf{T} : (v_0, E_0) :: \mathsf{equeuePush}(e_1, Q)$
    **case** $v_0 \prec v_1 = \mathsf{U} : (v_0, e_1 :: E_0) :: Q$
    **case** $v_0 \prec v_1 = \mathsf{F} : (v_1, \langle e_1\rangle) :: Q)$
```

`equeue_pop` $\vdash_{\mathrm{def}} \mathsf{equeuePop}(q_0 :: Q) = (q_0, Q)$

To use these definitions, it must be proven that they comply with the type definition, i. e. neither insertion nor removal destroy the order of the event queue but result to a valid event queue. This may be seen as an unnecessary overhead caused by the type definition, but the integration of the invariant in the type has the advantage that the conditions do not have to be mentioned in each definition or goal. With the help of these theorems, it is proven that pop and push operations maintain the order of the event queue.

`EQUEUE_PUSH_SORTED` $\vdash \mathsf{equeueSorted}(\mathsf{repQ}(\mathsf{equeuePush}(E, Q)))$
`EQUEUE_POP_SORTED` $\vdash \mathsf{equeueSorted}(\mathsf{repQ}(\mathsf{SND}(\mathsf{equeuePop}(Q))))$

The definition of the status structure is more complicated. In contrast to the event queue, the order is not invariant: It changes at each event. Thus, the compare function has three arguments; in addition to the two elements that are compared, the position where the comparison is made must be given. As the elements of the status structure are specific to the algorithm, only the signature is fixed, and the requirements for a total order are defined for a valid status structure order O.

`status_ref` $\vdash_{\mathrm{def}} \mathsf{statusRef}(O) = (O(p, s_1, s_1) = \mathsf{U})$
`status_antisym` $\vdash_{\mathrm{def}} \mathsf{statusAntisym}(O) = (O(p, s_1, s_2) = \ddot{\neg}\, O(p, s_2, s_1))$
`status_trans` $\vdash_{\mathrm{def}} \mathsf{statusTrans}(O) = (O(p, s_1, s_2) \ddot{*} O(p, s_2, s_3) \twoheadrightarrow O(p, s_1, s_3))$
`status_valid` $\vdash_{\mathrm{def}} \mathsf{statusValid}(O) = \mathsf{statusRef}(O) \wedge \mathsf{statusAntisym}(O) \wedge \mathsf{statusTrans}(O)$

The following definition states when a status structure is sorted with respect to a given order O at position p:

`status_sorted` $\vdash_{\mathrm{def}}$
$(\mathsf{statusSorted}(O, p, \langle\rangle) = \mathsf{T}) \wedge$
$(\mathsf{statusSorted}(O, p, \langle s\rangle) = \mathsf{T}) \wedge$
$(\mathsf{statusSorted}(O, p, s_1 :: s_2 :: S) = (O(p, s_1, s_2) = \mathsf{T}) \wedge \mathsf{statusSorted}(s_2 :: S))$

5.2.2 General Verification Approach

Since the main procedure of plane sweep is essentially the repeated handling of event points, it gives rise to a general proof strategy: With the help of invariants, the proof goals can be reduced to goals based on the handling of a single event point. Thus, similar advantages as for the implementation can be drawn from the plane sweep approach. The description of the general verification approach does not contain concrete definitions or theorems and is not very detailed, as it highly depends on the actual algorithm to verify.

Using an inductive argument, a correctness of an algorithm is divided in two parts: First, an invariant must be shown for the initial creation of the event queue. Then, the proof follows the plane sweep by verifying that each step of the plane sweep maintains the invariant.

A plane sweep algorithm does the following in each step: It takes the first element of the event queue and uses it to compute an event queue and a new (necessarily different) status structure that is well-ordered up to the next x-interval, i. e. between the current event point and the next one (which is the first one of the resulting event queue).

Naturally, the invariant used in the proof must imply the correctness of the algorithm. In general, it states that the constructed result left from the sweep line is correct. When the algorithm reaches the right end of the processed objects, the correctness for the whole plane can be deduced.

Finally, as the algorithm may add events in course of the plane sweep, the termination must also be shown explicitly.

5.3 Network Overlay

5.3.1 Problem

Networks are an important structure in computational geometry. They are the basis of the maps, which will be considered in Chapter 6. Before describing the overlay, the network is defined:

Definition 7 (Network). *A network consists of a pair $N = (vertices(N), edges(N))$, where:*

- *$vertices(N)$ is the set of vertices $vertices(N)$, which exactly consists of the endpoints of the edges:* $\mathsf{endpoints}(edges(\mathsf{N})) = vertices(N)$.
- *$edges(N)$ is the set of edges. They must be planar, i. e. free of any intersections. Two different edges may share only a common endpoint.*

The network overlay operation transforms a set of networks $\mathcal{N} = \{N_1, \ldots, N_n\}$ to a single network N by overlaying the edges of all networks in the following way: $vertices(N)$ contains all input vertices $\bigcup_{i=1\ldots n} vertices(N_i)$ as well as all the intersections of the edges. The set of edges $edges(N)$ is constructed by taking all input edges $\bigcup_{i=1\ldots n} edges(N_i)$ and then iteratively replacing all pairs of intersecting and overlapping edges with their overlays. Figure 32 shows a naive algorithm that represents a reference for the developed one: For all possible combinations of edges of the input networks, the overlay is computed. This operation is reiterated until no further intersections or overlaps are found.

Section 4.1.2 already showed how to compute the overlay of two line segments, and a slightly improved version of the naive approach is to take each pair of these segments and to check whether they intersect. If they intersect, they are replaced by their overlay. Since there are $O(n^2)$ of such

```
function NetworkOverlay($\mathcal{N}$)
    $N$ = newNetwork();
    $E = \bigcup_{N_i \in \mathcal{N}} edges(N_i)$;
    do
        $edges(N) = E$;
        $E = \{\}$;
        forall($e_1, e_2 \in edges(N)$)
            $E = E \cup$ segOverlayLines($e_1, e_2$);
    while($E = edges(N)$);
    $vertices(N) =$ endpoints($edges(N)$);
    return $N$;
```

Fig. 32. Naive Implementation of Network Overlay

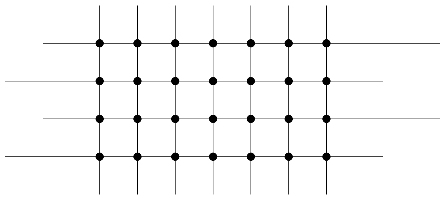

Fig. 33. $O(n^2)$ Intersections of n Line Segments

pairs, the algorithm needs $O(n^2)$ time. Theoretically, this approach is optimal, since each combination can result to an intersection. The example in Figure 33 illustrates this situation.

However, in most cases the number of actual intersections is relatively small compared to the number of possible ones. Hence, an algorithm that can exclude a lot of combinations before checking them, would be much faster in most cases. Thus, its complexity depends not only on the size of the inputs but also on the size of the output. Algorithms that show this behaviour are commonly called *output-sensitive*.

5.3.2 Algorithm

In the following, a solution to this problem is presented, which is based on the classic algorithm of Bentley and Ottmann [4]. This algorithm computes the intersection of n line segments with a plane sweep. The algorithm is extended and modified to compute the overlay by handling all degenerate cases with the help of the three-valued definitions of Chapter 4.

As a first version, consider the following algorithm: In the event queue, the left and right endpoints of each segment are stored, and the status structure contains all line segments that are intersected by the sweep line (theoretically in an arbitrary order, but here the segments are ordered by the y-coordinate of the intersection points). Only with this information, the number of test can be significantly reduced: Each time a new segment is inserted in the status structure, it is checked whether there are intersections with the other segments of the status structure. Generally, only a small number of the n segments are in the status structure at the same time, so that the average run-time can be reduced. For checking the correctness of the algorithm it must be verified that there can be no combination that results to an intersection and that the algorithm skips in the course of the computation. Considering Figure 34, it is seen that segments are at the same time in the status structure, if they have an overlapping x-interval, which is a necessary condition for the existence of an intersection. The proof can be derived from the bounding box check (see Section 4.4.3): An intersection point is in the x-intervals of both line segments — if a common interval does not exist, there cannot be an intersection point. Thus, the algorithm still computes all intersections, while reducing the number of intersections to be checked.

However, the algorithm can be still improved. In the first version, exclusions are performed only on the x-intervals of the segments. This approach can also be transformed in some way to y-intervals. When a new segment s_1 is inserted into the status structure, it is not needed to check for intersection with all other segments in the structure, but only with the neighbours. If an other, non-adjacent segment s_2 in the status structure has an intersection with s_1, all segments between them must either have an intersection point with s_1, s_2 or they end before the sweep line reaches the intersection between s_1 and s_2. In both cases, the segments s_1 and s_2 become eventually adjacent in the course of the plane sweep. For example, consider the segment s_5 in Figure 34. A check for an intersection with s_4 is not needed as long as s_2 is between both segments.

To implement this idea, the segments in the status structure must be sorted by the y-coordinate of their intersection point with the sweep line. Besides the insertion and removal of segments, intersection points change the order in the status structure. For this reason, they must be stored in the event queue, too. The general procedure is as follows: When a segment starts at the current event point, it is inserted into the status structure and it is checked for intersections with its neighbours. When a segment ends, it is removed from the status structure, and it is tested whether the two old neighbours of the segment (which have become adjacent) intersect. In both cases, if an intersection is detected, it is inserted into the event queue if it is on the right-hand side of the current event point.[3] When such an event is hit during the plane sweep, both segments are swapped in the status structure and are checked for intersections with their new neighbours. This is the general principle of the whole algorithm, which is often presented in textbooks. However, degenerated cases have not considered yet here.

To handle degenerate cases, some generalisations and modifications must be made to the original algorithm. Figure 35 shows an algorithm that considers all special cases, which were found while implementing the algorithm and reasoning about it with a theorem prover. In particular, the following situations have been considered:

[3] Note that this is necessary, since an intersection can be detected more than once. If the event had been already handled and was reinserted with a position left from the current event point, it would be processed twice — with incorrect results.

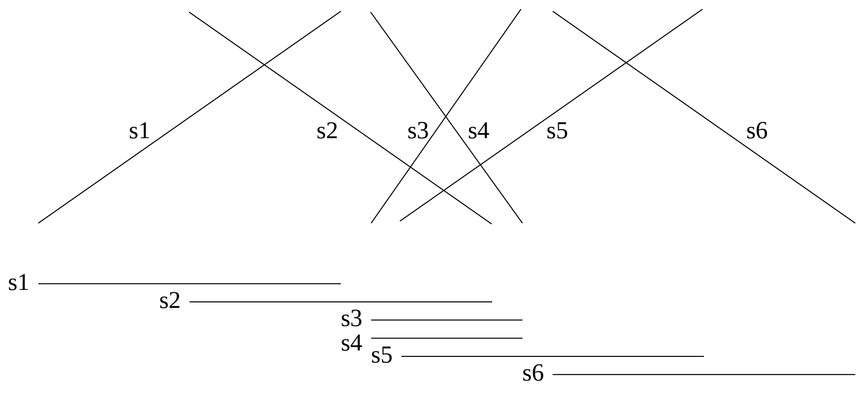

Fig. 34. Overlapping x-Intervals of Active Segments

```
function NetworkOverlay($e_1, \ldots, e_n$)
    $N$ = newNetwork();
    $Q$ = InitialiseEvents($e_1, \ldots, e_n$);
    $S = \langle\rangle$;
    while (not EventQueueEmpty($Q$))
        ($v_{\text{last}}, \mathcal{E}_{\text{begin}}, \mathcal{E}_{\text{end}}$) = EventQueuePop($Q$);
        if (SegLowmst($S$) $\neq \langle\rangle$)
            NetworkEdgesEnd();
        $S$ = StatusRemoveSet($S, \mathcal{E}_{\text{end}}$);
        $S$ = StatusInsertSet($S, \mathcal{E}_{\text{begin}}$);
        CheckIntersections($Q, S$);
        if (SegLowmst($S$) $\neq \langle\rangle$)
            NetworkEdgesBegin();
    return $N$;
```

Fig. 35. Network Overlay Plane Sweep

First, an arbitrary number of segments can intersect at an intersection point, not only two. Hence, the swapping of segment pairs cannot be done as described above, but all segments that intersect in the event point are reversed in the status structure. In the algorithm, this is emulated by removing and reinserting the segments to be reversed.

Second, events can have the same position, e. g. a segment can start at the point where another one ends. In this case, events are merged, and all parts must be handled. Although an arbitrary execution order is possible, ending segments are removed first, then intersecting ones are reversed, and finally starting ones are inserted. In this order, not only the status structure is kept as small as possible, but it also follows the topological order: Ending segments come from the left-hand side of the sweep line, intersecting ones can be found on both sides, while starting segments are on the right-hand side of the sweep line.

Third, two segments can intersect the sweep line at the same position. This is always the case for intersection event points. Thus, the order function of the status structure $\mathcal{S}$ must be refined. Segments are first ordered by the y-value of their intersection point with the current sweep line, then by their slope. Segments that are equal under both criteria are put in a list. Hence, the status structure stores a list of lists, where the elementary sublists contain collinear segments. These segments are ordered by the position of their right endpoints, which makes the detection of intersection points in CheckIntersection simpler - in principle, any other order would be generally possible. The last case for the order of the status structure are vertical segments: They are considered to be greater than any other segment, in the sublist sorted by the y-value of the right (i. e. upper) endpoint. The function StatusCompare implements all these rules.

Fourth, segments can touch or share common endpoints. These degenerated cases are handled without effort provided that the intersection check does not consider these cases as real intersections.

Before going into details about the network overlay algorithm, some helper functions are defined that are used in the following paragraphs. The most important ones are SegBelow(S), SegLowmst(S), SegUpmst(S), and SegAbove(S): SegBelow(S) returns the list of edges below the event point, SegAbove(S) the list of the edges above. These functions determine the position of the event point in the current status structure: SegLowmst(S) denotes the first list of edges

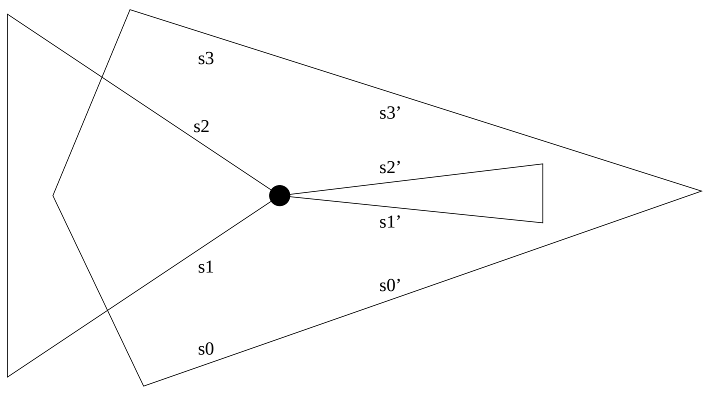

Fig. 36. Border Segments Functions

that contains the event point, and SegUpmst(S) is the last list. SegBelow(S) is the predecessor of SegLowmst(S), and SegAbove(S) is the successor of SegUpmst(S). If no corresponding list can be found, the empty list $\langle\rangle$ will be returned. Figure 36 illustrates the definitions. Just before the marked point, SegBelow$(S) = s_0$, SegLowmst$(S) = s_1$, SegUpmst$(S) = s_2$ and SegAbove$(S) = s_3$ hold — after the event point: SegBelow$(S) = s'_0$, SegLowmst$(S) = s'_1$, SegUpmst$(S) = s'_2$ and SegAbove$(S) = s'_3$ hold.

Event Queue

The left-hand side of Figure 37 shows the event queue functions. The event queue contains pairs $E = (v, (\mathcal{E}_1, \mathcal{E}_2))$ with the following meaning: The vector v denotes the position of the event. The sets $\mathcal{E}_1$ and $\mathcal{E}_2$ contain the edges that start and end at the event point, respectively. The function InsertEvent(Q, E) inserts the event E to the queue Q. The position of E in the queue is determined by its geometric position v (lexical order). If an event with the same geometric position already exists in the event queue, both events are merged by merging the edge sets. The event queue is initialised by InitialiseEvents by taking the edges of all input networks $M_1, \ldots, M_n$ and inserting their both endpoints to the event queue. When adding the segment to the event queue, the orientation of the edge must be considered so that the beginning of the edge is in front of its ending in the event queue.

The functions CheckIntersections and CheckIntersection, which are shown on the right-hand side of Figure 37, insert intersection events in the event queue during the plane sweep. The function CheckIntersections first determines which lists of the status structure must be checked. If there are segments that start at the current event point, the lowermost list passing the event point and the list below it are checked for intersections. Analogously, the uppermost list passing the event point and the list above are checked. If there are no segments that start at the event point, the lists above and below the event point become adjacent, and are checked for intersections. CheckIntersection distinguishes four cases: The first three ones cover the situations where one list or both of them do not exist. In the last case, the intersection point is computed. If it is an intersection point (neither a common segment nor a touching point), which is indicated by $t = \mathsf{T}$, all segments that contain the intersection point are added with an intersection event to queue.

```
function InsertEvent($\mathcal{Q}, (v, \mathcal{E}_1, \mathcal{E}_2)$)
switch
case $\mathcal{Q} = \langle\rangle$ :
   return $\langle E\rangle$;
case $\mathcal{Q} = q :: Q$ :
   $(v', \mathcal{E}_1', \mathcal{E}_2') = q$;
   switch
   case $v' \prec v$ :
      return $q$ :: InsertEvent($Q, (v, \mathcal{E}_1, \mathcal{E}_2)$);
   case $v' = v$ :
      return $(v, \mathcal{E}_1 \cup \mathcal{E}_1', \mathcal{E}_2 \cup \mathcal{E}_2') :: Q$;
   case $v' \succ v$ :
      return $(v, \mathcal{E}_1, \mathcal{E}_2) :: q :: Q$;

function InitialiseEvents($N_1, \ldots, N_n$)
   $Q := \langle\rangle$;
   forall $(i \in \{1 \ldots n\})$
      forall $(e \in edges(N_i))$
         if $(src(e) \prec dest(e) = \mathsf{T})$
            $Q$ = InsertEvent($Q, (src(e), \{e\}, \{\})$);
            $Q$ = InsertEvent($Q, (dest(e), \{\}, \{e\})$);
         else
            $Q$ = InsertEvent($Q, (dest(e), \{e\}, \{\})$);
            $Q$ = InsertEvent($Q, (src(e), \{\}, \{e\})$);
   return $Q$;
```

```
function CheckIntersections($Q, S$)
   if (SegLowmst($S$) $\neq \langle\rangle$)
      CheckIntersection
         ($Q$, SegBelow($S$), SegLowmst($S$));
      CheckIntersection
         ($Q$, SegUpmst($S$), SegAbove($S$));
   else
      CheckIntersection
         ($Q$, SegBelow($S$), SegAbove($S$));

function CheckIntersection($Q, \mathcal{S}_1, \mathcal{S}_2$)
switch
case $(\mathcal{S}_1, \mathcal{S}_2) = (\langle\rangle, \langle\rangle)$ :
   return $Q$;
case $(\mathcal{S}_1, \mathcal{S}_2) = (s_1 :: S_1, \langle\rangle)$ :
   return $Q$;
case $(\mathcal{S}_1, \mathcal{S}_2) = (\langle\rangle, s_2 :: S_2)$ :
   return $Q$;
case $(\mathcal{S}_1, \mathcal{S}_2) = (s_1 :: S_1, s_2 :: S_2)$
   $(t, v_1, v_2)$ = StatusIntersection($s_1, s_2$);
   if $(t = \mathsf{T})$
      forall($e \in s_1, s_2$)
         if (onEdge($v, e$) $= \mathsf{T}$)
            $Q$ = InsertEvent($Q, (v_1, \{e\}, \{e\})$);
   return $Q$;
```

Fig. 37. Network Overlay Algorithm: Event Queue and Intersection Detection

Status Structure

Figure 38 shows the functions for the status structure. Its core structure is the order function StatusCompare(p, e_1, e_2). It compares two edges e_1 and e_2 at the position p according to the order described above. StatusInsertSet($S, \mathcal{E}$) inserts all edges of the set $\mathcal{E}$ in the status structure, which is implemented by StatusInsert(S, e): In the case that two segments are equal with respect to the order function, they are put in the same sublist ordered by their right endpoints. With this ordering, the intersection for collinear line segments can be implemented more efficiently. The removal of segments is handled analogously by StatusRemoveSet($S, \mathcal{E}$) and StatusRemove(S, e).

The handling of an event point is divided into three steps: First, the function NetworkEdgesEnd() processes all edges that end at the current event point or pass it. These edges are determined by SegLowmst(S) and SegUpmst(S). For each edge list s, a new edge in the output network is created from the startpoint stored in $curStart(s)$ to the current event point v_{last}. This point is set as the startpoint for all edge lists that start at this event point in the function NetworkEdgesBegin().

```
function StatusInsertSet($\mathcal{S}, \mathcal{E}$)
    forall ($e \in \mathcal{E}$)
        $S$ = StatusInsert($\mathcal{S}, e$);
    return $S$;

function StatusInsert($\mathcal{S}, e$)
    switch
    case $\mathcal{S} = \langle\rangle$ :
        return $\langle\langle e\rangle\rangle$;
    case $\mathcal{S} = s :: S$ :
        switch
        case StatusCompare($v_{\text{last}}, s, e$) = T :
            return $s ::$ StatusInsert($S, e$);
        case StatusCompare($v_{\text{last}}, s, e$) = U :
            return StatusCollinear($s, e$) $:: S$;
        case StatusCompare($v_{\text{last}}, s, e$) : F :
            return $\langle e\rangle :: s :: S$;

function StatusCollinear($s :: S, e$)
    if (RightEnd($e$) $\prec$ RightEnd($s$) = T)
        return $s ::$ StatusCollinear($S, e$);
    else
        return $s :: S$;

function StatusCompare($p, e_1, e_2$)
    $y_1$ = yValueAt($e_1$, X($p$));
    $y_2$ = yValueAt($e_2$, X($p$));
    if ($y_1$ = U and $y_2$ = U)
        return U;
    if ($y_1$ = U and $y_2 \neq$ U)
        return ($y_2 <$ Y($p$) $\Rightarrow$ T| F);
    if ($y_1 \neq$ U and $y_2$ = U)
        return ($y_2 <$ Y($p$) $\Rightarrow$ F| T);
    if ($y_1 \neq y_2$)
        return ($y_1 < y_2 \Rightarrow$ T| F);
    $\delta_1$ = slope($e_1$);
    $\delta_2$ = slope($e_2$);
    if ($y_1 <$ Y($p$))
        return ($\delta_2 < \delta_1 \Rightarrow$ T| F);
    else
        return ($\delta_1 < \delta_2 \Rightarrow$ T| F);
    return U;

function NetworkEdgesEnd()
    for ($s$ = SegLowmst($S$) ... SegUpmst($S$))
        $e_{\text{last}}$ = newEdge();
        $edges(N) = edges(N) \cup \{e_{\text{last}}\}$;
        $src(e_{\text{last}}) = curStart(s)$;
        $dest(e_{\text{last}}) = v_{\text{last}}$;

function NetworkEdgesBegin()
    for ($s$ = SegLowmst($S$) ... SegUpmst($S$))
        $curStart(s) = v_{\text{last}}$;
```

Fig. 38. Network Overlay: Status Structure and Edge Handling

5.4 Network Overlay Verification

The verification of the network overlay algorithm is based on the general definitions of Section 5.2.2. In addition to the general descriptions, the formalisations are specialised for the network overlay.

The description of the algorithm is taken from the previous section with some exceptions: Sequences and loops, which are not available in a functional style, are eliminated by substitution and recursion. Lists are used instead of sets, since they are more convenient for proofs. Vertices are seen to be equivalent to points, and line segments are substituted for edges.

5.4.1 Event Queue

The first step in the verification is the formalisation of the event queue initialisation:

```
NetworkInitialiseEvents $\vdash_{\text{def}}$
    $(\mathsf{InitialiseEvents}(\mathcal{Q}, \langle\rangle) = \mathcal{Q}) \wedge$
    $(\mathsf{InitialiseEvents}(\mathcal{Q}, l :: \mathcal{L}) =$ if $\mathsf{beg}(l) \prec \mathsf{end}(l) = \mathsf{T}$ then
            $\mathsf{equeuePush}(\mathsf{equeuePush}(\mathsf{InitialiseEvents}(\mathcal{Q}, \mathcal{L}),$
                $\mathsf{end}(l), (\langle\rangle, \langle l\rangle)),\ \mathsf{beg}(l), (\langle l\rangle, \langle\rangle))$
        else
            $\mathsf{equeuePush}(\mathsf{equeuePush}(\mathsf{InitialiseEvents}(\mathcal{Q}, \mathcal{L}),$
                $\mathsf{beg}(l), (\langle\rangle, \langle l\rangle)),\ \mathsf{end}(l), (\langle l\rangle, \langle\rangle))$
```

In addition to the general restrictions that apply to the content of the event queue, the following properties must hold for the network overlay event queue: For each startpoint of an edge, there is exactly one endpoint at a position that is greater. Moreover, intersection events only occur between the start and the endpoint of a segment (which is encoded by first removing and then inserting new segments).

```
ovQueueValid $\vdash_{\text{def}}$
    $(\mathsf{ovQueueValid}(\langle\rangle, L) = (L = \{\})) \wedge$
    $(\mathsf{ovQueueValid}((v, (\langle\rangle, \langle\rangle)) :: \mathcal{Q}, L) = \mathsf{ovQueueValid}(\mathcal{Q}, \mathcal{L})) \wedge$
    $(\mathsf{ovQueueValid}((v, (l : \mathcal{L}_1, \langle\rangle)) :: \mathcal{Q}, L) =$
        if $l \in L$ then $\mathsf{F}$ else $\mathsf{ovQueueValid}((v, (\mathcal{L}_1, \langle\rangle)) :: \mathcal{Q}, L \cup \{l\})) \wedge$
    $(\mathsf{ovQueueValid}((v, (\mathcal{L}_1, l : \mathcal{L}_2)) :: \mathcal{Q}, L) =$
        if $l \notin L$ then $\mathsf{F}$ else $\mathsf{ovQueueValid}((v, (\mathcal{L}_1, \mathcal{L}_2)) :: \mathcal{Q}, L \setminus \{l\}))$
```

It is proven that the initial event queue is valid and that a single step in the network overlay algorithm maintains the validity. This property is needed as an invariant for the correctness proof.

5.4.2 Status Structure

As the first step, the status structure ordering is formalised according to the definition in Figure 38. However, two details differ from the algorithm:

The status structure only contains single segments and not lists of segments to keep the proof goals manageable, which are already hard to view as a whole. This involves a modification of the insert and remove operation: When inserting, the maximum of both segments is taken, and removing is only done if the maximum position is reached at the event point.

```
status_insert $\vdash_{\text{def}}$ $\mathsf{statusInsert}(p, s_1, \langle\rangle) = \langle s_1\rangle \wedge$
                    $\mathsf{statusInsert}(p, s_1, s_0 :: S) =$
                        switch3
                            $(\mathsf{StatusCompare}(p, s_0, s_1),$
                                $(s_0 :: \mathsf{statusInsert}(p, s_1, S)),$
                                $((\mathsf{mkLine}(\mathsf{beg}(s_0), \mathsf{vMax}(\mathsf{end}(s_0), \mathsf{end}(s_1)))) :: S),$
                                $(s_1 :: s_0 :: S)$
                        )
```

Second, the order is defined in two steps: First, the y-coordinates of the segments are compared and then the slope. This has some advantages for the later proof, since later formalisations can state the fact that a segment passes an event point. With regard to the statusCompareY, all segment in an event point are equal, but they may differ with regard to statusCompareSlope. Furthermore, statusValid is proven to be a valid order as defined in Section 5.2.

$$
\begin{array}{l}
\texttt{statusCompare} \vdash_{\text{def}} \mathsf{StatusCompare}(p, s_1, s_2) = \\
\quad \textbf{if } \mathsf{statusCompareY}(p, s_1, s_2) \neq \mathsf{U} \\
\quad\; \textbf{then } \mathsf{statusCompareY}(p, s_1, s_2) \\
\quad\; \textbf{else } \mathsf{statusCompareSlope}(p, s_1, s_2) \\
\texttt{OV\_STATUS\_VALID} \vdash \mathsf{statusValid}(\mathsf{StatusCompare})
\end{array}
$$

The next step is to prove that the status compare function $\mathsf{StatusCompare}(p, l_1, l_2)$ changes at an intersection p of two line segments l_1 and l_2. Based on this theorem, it is proven that a constant StatusCompare implies the non-existence of an intersection point between both points. For this theorem, the uniqueness of the intersection point is used in the course of the proof.

$$
\begin{array}{l}
\texttt{OV\_STATUS\_COMPARE\_CHANGE} \vdash \\
\quad \mathsf{isLineIntersect}(p, l_1, l_2) \implies \\
\quad (p_1 \prec p = \mathsf{T}) \implies (p \prec p_2 = \mathsf{T}) \implies \\
\quad (\mathsf{StatusCompare}(p, l_1, l_2) = \ddot{\neg}\, \mathsf{StatusCompare}(p, l_1, l_2)) \\
\texttt{OV\_STATUS\_COMPARE\_CONSTANT} \vdash \\
\quad \mathsf{isLineIntersect}(p, l_1, l_2) \implies \\
\quad (p_1 \prec p_2 = \mathsf{T}) \implies \\
\quad (\mathsf{StatusCompare}(p_1, l_1, l_2) = \mathsf{StatusCompare}(p_2, l_1, l_2)) \implies \\
\quad ((p \prec p_1 = \mathsf{T}) \vee (p_2 \prec p = \mathsf{T}))
\end{array}
$$

Another function that is needed for the prove is definition of the last event point for a given status structure. Since right endpoints of segments cannot be reconstructed from the current status structure, it gives the last insertion or intersection point (which is equivalent due to the encoding of the intersection points). For an empty status structure, the function remains undefined.

$$
\begin{array}{l}
\texttt{statusLast} \vdash_{\text{def}} \\
\quad (\mathsf{statusLast}(\langle s_1 \rangle) = \mathsf{beg}(s_1)) \wedge \\
\quad (\mathsf{statusLast}(s_1 :: s_2 :: S) = \mathsf{vMax}(\mathsf{beg}(s_1), \mathsf{statusLast}(s_2 :: S)))
\end{array}
$$

5.4.3 Algorithm

The termination of the network overlay algorithm is not obvious to the theorem prover, as it is not a primitive recursive function over the event queue: New elements may be inserted in the event queue in each step. To avoid this problem, the verification makes a modified induction over the actual event queue of the network overlay algorithm. In this proof-friendly version, all events are computed in advance, i. e. the event queue initially contains events for all endpoints as well as all intersections, ordered according to their position. The modified event queue has a big advantage: New elements do not need to be inserted in the course of the plane sweep and thus, termination of the plane sweep does not need to be proven.

Instead of directly using an induction to prove the goal, the crucial property of the network overlay is analysed first: It must be checked that the algorithm really detects all intersections. As the event queue contains all intersections by construction, there cannot be any intersections between two events. It must be checked whether all intersections have been added by the network overlay before, i. e. that the two intersecting segments have been neighbours at a previous event point. Since intersection points are not explicitly stored in the event queue, this property is checked for all segment pairs that end at a common event point. These points are clearly a superset of the intersection points, as intersections are encoded by an ending and a starting of the respective segments.

$$
\begin{aligned}
&\texttt{statusNeighbour} \vdash_{\text{def}}\\
&\quad (\mathsf{statusNeighbour}(\langle\rangle, s_a, s_b) = \mathsf{F})\land\\
&\quad (\mathsf{statusNeighbour}(\langle s_1\rangle, s_a, s_b) = \mathsf{F})\land\\
&\quad (\mathsf{statusNeighbour}(s_1 :: s_2 :: S, s_a, s_b) =\\
&\qquad (s_1 = s_a) \land (s_2 = s_b) \lor (s_1 = s_b) \land (s_2 = s_a) \lor \mathsf{statusNeighbour}(s_2 :: S, s_a, s_b)
\end{aligned}
$$

Assume that the algorithm is at an intersection event point for two segments l_1 and l_2. In principle, it is enough to look at the previous event point. This is very convenient, as it does not require an additional induction step. Two cases can then be distinguished: If this point was a point, where only segments end, both segments were tested for an intersection, as they evidently became neighbours at that event point.

If a segment starts there, it must be still between them, as there cannot be an intersection in the x-interval in between. At the current event point, all points intersect as stated by the following theorem: If two non-parallel segments have the same y-value at an event point, this point is the intersection point. Moreover, the segment cannot be parallel, since they had different y-values at the previous event point.

$$
\begin{aligned}
&\texttt{YVALUE\_EQUAL\_INTERSECT} \vdash\\
&\quad \neg\mathsf{parallel}(l_1, l_2) \implies\\
&\quad \mathsf{statusCompareY}(p, s_1, s_2) = \mathsf{U} \implies\\
&\quad \mathsf{isLineIntersect}(p, l_1, l_2)
\end{aligned}
$$

Finally, there is a third case: The last event point cannot be between the two segments. Then, it would be irrelevant for the detection of the intersection and the point before must be considered, and possibly the one before that one, and so on. To avoid an backward induction of the event points (which is difficult), a stripping function is defined, which removes event points outside the considered segments. statusStrip returns the status structure without the irrelevant segments, whereas statusStripped returns the stripped elements.

$$
\begin{aligned}
&\texttt{statusStrip} \vdash_{\text{def}}\\
&\quad (\mathsf{statusStrip}(\langle\rangle, s_a, s_b, v) = \langle\rangle)\land\\
&\quad (\mathsf{statusStrip}(s_1 :: S, s_a, s_b, v) =\\
&\qquad \textbf{if } (\mathsf{StatusCompare}(v, s_a, s_1) = \mathsf{T}) \lor (\mathsf{StatusCompare}(v, s_1, s_b) = \mathsf{T})\\
&\qquad \textbf{then } s_1 :: (\mathsf{statusStrip}(S, s_a, s_b, v))\\
&\qquad \textbf{else } (\mathsf{statusStrip}(S, s_a, s_b, v))\\
&\texttt{statusStripped} \vdash_{\text{def}}\\
&\quad (\mathsf{statusStripped}(\langle\rangle, s_a, s_b, v) = \{\})\land\\
&\quad (\mathsf{statusStripped}(s_1 :: S, s_a, s_b, v) =\\
&\qquad \textbf{if } (\mathsf{StatusCompare}(v, s_a, s_1) = \mathsf{T}) \lor (\mathsf{StatusCompare}(v, s_1, s_b) = \mathsf{T})\\
&\qquad \textbf{then } s_1 \cup (\mathsf{statusStripped}(S, s_a, s_b, v))\\
&\qquad \textbf{else } (\mathsf{statusStripped}(S, s_a, s_b, v))
\end{aligned}
$$

The strip function must be proven not to remove relevant intersection test as shown by the following theorem:

$$
\begin{aligned}
&\texttt{OV\_STRIPPED\_CORRECT} \vdash \\
&\quad s \in \mathsf{statusStripped}(S, s_a, s_b, v) \implies \\
&\quad v_0 = \mathsf{statusLast}(S) \implies \\
&\quad \neg\exists v.\, v_0 \prec v \wedge (\mathsf{isLineIntersect}(v, s, s_a) \vee \mathsf{isLineIntersect}(v, s, s_b))
\end{aligned}
$$

If this strip function is applied to the above situation, the third case cannot occur and the network overlay algorithm is guaranteed to detect all line intersections.

5.5 Implementation Details

5.5.1 Intersection Test Reduction

The number of intersection tests can be significantly reduced by the following two considerations:

First, input networks can be assumed to be planar. For each element in the status structure, it is stored, which maps contribute to the segment. As each element may be built from several overlapping edges of different input maps, a list of maps is associated with each entry. Before an intersection check, these lists are compared: If they are equal, the intersection test must fail due to the planarity of the input networks.

Second, a bounding-box check is used for a fast line segment intersection test. The plane sweep approach inherently contains a half of the bounding-box check. As already discussed, only segments with overlapping x-intervals are checked. The other half can be integrated as follows: Before the actual intersection test, it is checked whether the respective two segments have a common y-interval. Since the vertical ordering of the segments can be derived from status structure, the algorithm only needs to determine the upper vertex of the lower segment v_1 and the lower vertex of the upper segments v_2 and check whether $\mathsf{above}(v_1, v_2) \geq \mathsf{U}$. Only if this test is successful, the actual intersection check is needed.

5.5.2 Event Queue Ordering

Algorithms that follow the plane sweep approach first construct an event queue. As all events are ordered, this involves a sorting of all the events induced by the inputs. Thus, for m networks containing each $O(n)$ vertices, the asymptotic complexity of the event queue construction in the network overlay is generally $O(n \cdot m \cdot \log(n \cdot m))$.

However, if the input is a construction by a previous network overlay algorithm, this fact can be exploited. The network overlay algorithm constructs the vertices in lexical order. So, the vertices are stored in a list instead of a set to keep their order. With the help of this property, the constructing the initial event queue can be done in $O(n \cdot m \cdot \log m)$.

Furthermore, when inserting an edge in the event queue, its orientation must be normalised that the overlay algorithm can assume that the destination of an edge is its right endpoint. When edges are guaranteed to have this order by construction, the check can be skipped and the edge is inserted without a comparison.

For the polygon-processing library, the theoretical complexity is not the significant number. However, tests have shown that these modifications speed up the run-time of the overlay algorithm by 5 – 40 % depending on the number of vertices and networks. Some care must be taken when networks are created from scratch. Their vertices must be sorted. Otherwise, the overlay algorithm returns (if the inputs are not checked) a completely useless result.

Usually, trees or similar data structures are used for the event queue and the status structure of the plane sweep, since they guarantee a good asymptotic complexity. For the problem sizes usually considered for the polygon-processing library (about 100–200 segments per overlay), simple double-linked lists have shown to be a bit more efficient. Moreover, they are better with respect to the memory usage, which will be explained in the following subsection.

5.5.3 Static Memory

A special requirement for many embedded systems is the restriction to static memory. This requirement is due to several reasons: First, dynamic memory libraries may not be available for the embedded device. Second, allocation of memory is always a source of bugs. Memory leaks are among the most frequent programming errors, which should be avoided for safety-critical systems. Third, only statically reserved memory may be efficient enough. In particular, this is the case if real-time requirements must be met, where the size of the data structures must be bounded a priori to give a guaranteed worst-case execution time.

All data structures for the plane sweep algorithm are implemented with a fixed upper bound of memory usage. Clearly, this implies restrictions to the size of the inputs, which will be discussed in detail in Section 6.5.

Chapter 6

Polygons and Maps

So far, only one-dimensional objects like lines, segments and their intersections have been studied. Based on them, this chapter introduces polygons and maps as the main two-dimensional structures, which are the core of the developed polygon-processing library. They are able to describe areas in the plane, which may be intepreted differently by single applications.

First, Section 6.1 defines polygons and their interior. The more general concept of maps will be discussed in Section 6. Based on this data structure, the overlay of a set of maps is presented in Section 6.3. Sections 6.4 and 6.5 show how the algorithm can be adapted to the needs of embedded systems: Conservative rounding and map simplification limit the problem size in a well-defined way so that real-time requirements can be met.

6.1 Polygons

6.1.1 Boundary

A general polygon consists of a number of contours, where each one is usually defined by a sequence of vertices $\mathcal{V} = \langle v_1, \dots, v_n \rangle$. The boundary then consists of the edges that connect these vertices, where consecutive edges share a common endpoint: $\mathcal{E} = \langle e_1, \dots, e_n \rangle$, where e_i connects v_i and v_{i+1} for $1 \leq i < n$, and e_n connects v_n and v_1.

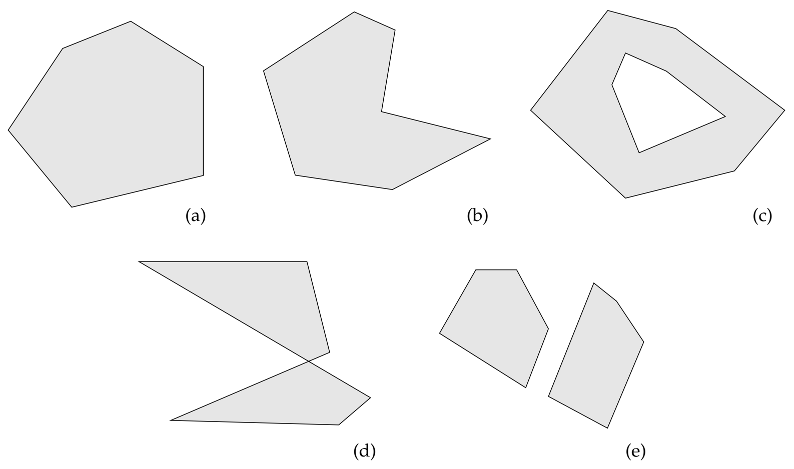

Fig. 39. Different Kinds of Polygons: Convex, Concave, With Hole, Self-Intersecting, Multi-Contour

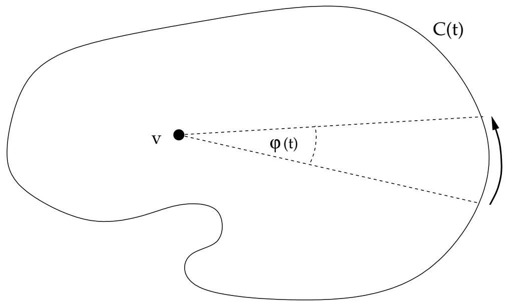

Fig. 40. Winding Number Definition

Figure 39 shows several examples of polygons. Polygons can be convex (a), concave (b) or contain holes (c). Contours can be self-intersecting as in polygon (d), or a polygon can consist of several unconnected contours.

6.1.2 Interior of a Polygon

The general definition of the interior of a polygon is not straightforward. There are a lot of anomalies that have to be considered when the interior is defined: contours of the polygon may be self-intersecting, consecutive vertices may be identical, or some edges may overlap. A practical way to cope with all these issues is the definition of a winding number of a point p with regard to a contour C [80]: It is given by the integrated angle φ and basically describes how often a point winds around the point p when following the contour C (see Figure 40):

Definition 8 (Winding Number). *For a contour C in parametric form*

$$c : [a, b] \mapsto C \subset \mathbb{R}^2 \textit{ with } c(t) = \begin{pmatrix} x(t) \\ y(t) \end{pmatrix}$$

and a point p, which is not on the contour, the winding number $w_C(p)$ is the oriented line integral of the angle $\varphi(t) = \arctan \frac{y(t)-\mathsf{Y}(p)}{x(t)-\mathsf{X}(p)}$:

$$\omega_C(p) = \frac{1}{2\pi} \int \frac{d\varphi}{dt}(t)\, dt = \frac{1}{2\pi} \int \frac{\dot{y}(t)x(t) - y(t)\dot{x}(t)}{x(t)^2 + y(t)^2}\, dt$$

For closed contours, the winding number associates an integer to all points that are not on a contour. Based on this number, the interior can be defined: A usual choice is to define all points that have an odd winding number as inside, whereas other choices (positive, non-zero) are also possible. Figure 41 shows three exemplary polygons with winding numbers.

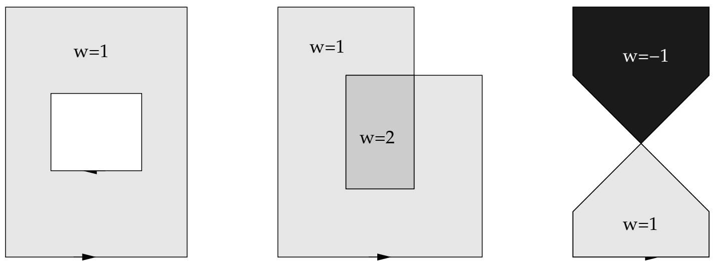

Fig. 41. Winding Number Examples

The usual choice of selecting all points with an odd winding number corresponds to the odd-even rule that was implemented by the parity algorithm in Section 4.3.1. This algorithm also complies with the three-valued setting of the geometry kernel by assigning U to all points on the edges of a polygon.

In the following, polygons will not be considered anymore, since they can be subsumed by more general structures: in particular by maps. Since these maps are planar by definition, they are guaranteed to be free of all the mentioned anomalies, which is an optimal setting for algorithms based on them.

6.2 Maps

As the name suggests, maps and the map overlay problem have their origin in the domain of geographic information systems. In polygon-processing and situation analysis applications, they can be used to describe polygonal regions or objects, e. g. vehicles, obstacles, dangerous or safe areas. Although maps are planar by definition, they are able to store a set of (possibly overlapping) polygons by using various labellings.

In general, there are two kinds of maps: bounded and unbounded maps. In unbounded maps, edges with an infinite length are possible, e. g. a line that separates the plane into two subdivisions. In contrast to this, bounded maps only contain finite edges, which are all fixed by two endpoints. Although all algorithms in the following can be extended to unbounded maps, unbounded maps are not considered here. Since they are not relevant for the considered embedded applications, unbounded maps would unnecessarily increase the complexity.

Definition 9 (Map). *A bounded map in the plane consists of a triple $M = (\mathcal{V}, \mathcal{E}, \mathcal{F})$, where:*

- $\mathcal{V} = \mathit{vertices}(M) \subseteq \mathbb{Q}^2$ *is a finite set of* vertices.
- $\mathcal{E} = \mathit{edges}(M) \subseteq \mathcal{V} \times \mathcal{V}$ *is the set of* edges. *An edge $e = (v_i, v_j)$ connects two different vertices $v_i, v_j \in \mathcal{V}$. A point v can belong to more than one line segment induced by an edge $e = (v_i, v_j)$ only if it is one of the vertices v_i or v_j. Since edges do therefore not intersect, a map is always a planar graph.*
- *The edges of a map induce a partition of the plane into a set of* faces $\mathcal{F} = \mathit{faces}(M)$. *Each face is a polygonal region bounded by one outer component (sequence of edges) and possibly several inner components, which are commonly referred to as holes. Moreover, a map has exactly one unbounded face $\mathit{unbdFace}(M)$, which has only inner components.*

In the following, all faces are assumed to be labelled. The multiset $\mathit{labels}(f)$ associated with a face $f \in \mathcal{F}$ contains the attribute information, i. e. it describes the properties of f on application level. For instance, this information can describe the membership to a polygon.
Figure 42 gives an example map and illustrates the data structure, which is used in the algorithms. Maps are represented as doubly connected edge lists: In this data structure, edges consist of pairs of half-edges representing both directions of an edge. Two half-edges forming a pair are called twins, and each one has a references to the adjacent face. For the sake of simplicity, this detail is hidden in the following, and only edges are used in the descriptions of the algorithms.

Thus, each edge e has references (dashed lines) to the two vertices $\mathit{src}(e)$ and $\mathit{dest}(e)$ that it connects. The face that lies on its left-hand side (as seen from source to destination) is its $\mathit{leftFace}(e)$, the opposite one is its $\mathit{rightFace}(e)$. A face f has references to its outer component

$outerComp(f)$ and the set of its inner components $innerComp(f)$. They point to an arbitrary edge of the component, from where the whole component can be traversed. Finally, each map M has a reference to its unbounded face $unbdFace(M)$.

6.3 Map Overlay

6.3.1 Problem

In the previous chapter, the overlay for a network was computed. In this section, an algorithm for the analogous operation on maps is described. The map overlay operation can be used to combine a set of inputs maps (sometimes called layers) into a single map. It covers a lot of other geometric problems by means of simple reductions (see Section 6.3.3). It is defined as follows:

Definition 10. *The overlay of a set of maps $\mathcal{M} = \{M_1, \ldots, M_n\}$ where each map is given by $M_i = (vertices(M_i), edges(M_i), faces(M_i))$ is the map $M = (vertices(M), edges(M), faces(M))$:*

- *The set of vertices $vertices(M)$ and the set of edges $edges(M)$ correspond the result of the network overlay of the maps in $\mathcal{M}$.*
- *The set of faces $faces(M)$ is induced by the set of edges $edges(M)$ of the new map, as in the definition of a single map.*

The labels $labels(f)$ for a face $f \in faces(M)$ of the overlay is defined as expected: Each face is labelled with the sum of labels of the faces that contribute to this face. To determine the set of contributing faces, choose an arbitrary point inside the face and locate in each input layer the face that contains it.

Since the significant information is the labelling of the faces, redundant vertices and edges are removed after a map overlay: Edges are redundant if the faces on both of its sides have the same labelling. Consecutively, vertices are removed if no edge is connected to them.

6.3.2 Algorithm

The network overlay algorithm, which is presented in Section 5.3 is the basis of the map overlay algorithm. It also copes with degenerate inputs and thus, it handles all maps. Again, the handling

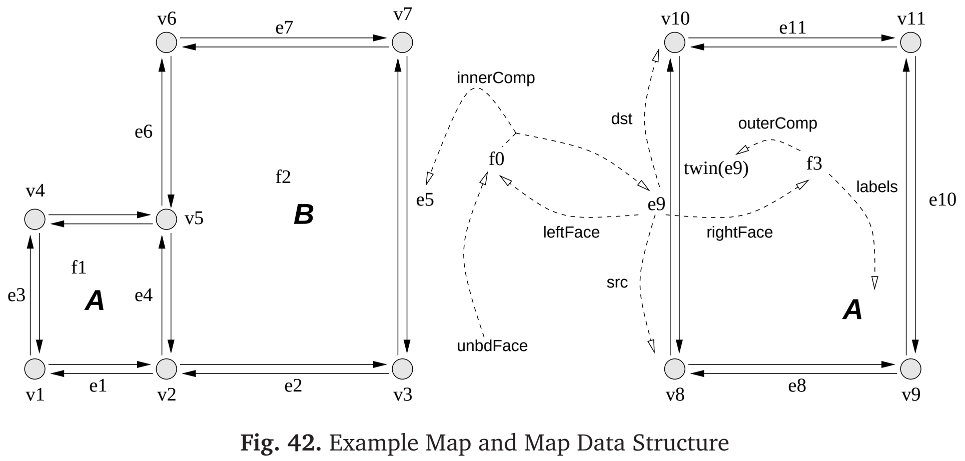

Fig. 42. Example Map and Map Data Structure

```
function MapOverlay($M_1, \dots, M_n$)
    $M$ = newMap();
    $curFace(\perp) = unbdFace(M)$;
    $curRaw(\perp) = \sum_{i=1}^{n} labels(unbdFace(M_i))$;
    $curLabels(\perp)$ = computeLabels($curRaw(\perp)$);
    $labels(unbdFace(M)) = curLabels(\perp)$;
    InitialiseEvents($M_1, \dots, M_n$);
    $S = \langle\rangle$;
    while ($Q \neq \langle\rangle$)
        $(v_{\text{last}}, \mathcal{E}_{\text{begin}}, \mathcal{E}_{\text{end}})$ = head($Q$);
        $vertices(M) = vertices(M) \cup \{v_{\text{last}}\}$;
        $Q$ = tail($Q$);
        $f_{\text{above}} = curFace$(prev(SegAbove($S$)));
        $f_{\text{below}} = curFace$(SegBelow($S$));
        $n_{\text{end}} = 0$;
        $n_{\text{begin}} = 0$;
        if (SegLowmst($S$) $\neq \perp$)
            HandleEdgesEnd();
        $S$ = StatusRemoveSet($S, \mathcal{E}_{\text{end}}$);
        $S$ = StatusInsertSet($S, \mathcal{E}_{\text{begin}}$);
        CheckIntersections($Q, S$);
        if (SegLowmst($S$) $\neq \perp$)
            HandleEdgesBegin();
        HandleMerge();
        if ($n_{\text{end}} = 0$ and $n_{\text{begin}} = 0$)
            $vertices(M) = vertices(M) \setminus \{v_{\text{last}}\}$;
    $outerComp(unbdFace(M)) = \perp$;
    return $M$;
```

Fig. 43. Map Overlay Algorithm

of degeneracies is by far the most tricky part of the algorithm. While implementing the algorithm a lot of imprecise explanations and forgotten cases in other descriptions have been found, although many authors claim to care about degenerate cases: e. g. the authors of [36] do not handle overlapping edges in the map overlay algorithm.

Instead of removing redundant parts after the overlay, the presented algorithm only constructs the significant parts with respect to a given operation on the labeling: Redundant vertices and edges (with respect to the labeling), isolated vertices are immediately removed in the course of the plane sweep, and new edges are only created if the labels of the faces on both sides are different. This optimisation leads to a remarkably better performance while additionally decreasing the memory usage in the course of the algorithm.

The main loop of the algorithm is shown in Figure 43: For the input maps M_1 to M_n, the algorithm computes the overlay M. It basically follows the structure of the network overlay algorithm presented in Section 5.3. Event queue and status structure used there are also the basis of the map overlay algorithm.

In addition to the border segment functions (SegBelow(s) and others), the algorithm keeps track of the face (of the output map M) that is below and above the current event point. As this

```
function InsertEvent($\mathcal{Q}, (v, \mathcal{E}_1, \mathcal{E}_2)$)
switch
case $\mathcal{Q} = \langle\rangle$ :
   return $\langle E \rangle$;
case $\mathcal{Q} = q :: Q$ :
   $(v', \mathcal{E}'_1, \mathcal{E}'_2) = q$;
   switch
   case $v' \prec v$ :
      return $q$ :: InsertEvent($Q, (v, \mathcal{E}_1, \mathcal{E}_2)$);
   case $v' = v$ :
      return $(v, \mathcal{E}_1 \cup \mathcal{E}'_1, \mathcal{E}_2 \cup \mathcal{E}'_2) :: Q$;
   case $v' \succ v$ :
      return $(v, \mathcal{E}_1, \mathcal{E}_2) :: q :: Q$;

function InitialiseEvents($N_1, \ldots, N_n$)
   $Q := \langle\rangle$;
   forall $(i \in \{1 \ldots n\})$
      forall $(e \in edges(N_i))$
         if $(src(e) \prec dest(e) = \mathsf{T})$
            $Q$ = InsertEvent($Q, (src(e), \{e\}, \{\})$);
            $Q$ = InsertEvent($Q, (dest(e), \{\}, \{e\})$);
         else
            $Q$ = InsertEvent($Q, (dest(e), \{e\}, \{\})$);
            $Q$ = InsertEvent($Q, (src(e), \{\}, \{e\})$);
   return $Q$;

function CheckIntersections($Q, S$)
   if (SegLowmst($S$) $\neq \langle\rangle$)
      CheckIntersection
         ($Q$, SegBelow($S$), SegLowmst($S$));
      CheckIntersection
         ($Q$, SegUpmst($S$), SegAbove($S$));
   else
      CheckIntersection
         ($Q$, SegBelow($S$), SegAbove($S$));

function CheckIntersection($Q, \mathcal{S}_1, \mathcal{S}_2$)
switch
case $(\mathcal{S}_1, \mathcal{S}_2) = (\langle\rangle, \langle\rangle)$ :
   return $Q$;
case $(\mathcal{S}_1, \mathcal{S}_2) = (s_1 :: S_1, \langle\rangle)$ :
   return $Q$;
case $(\mathcal{S}_1, \mathcal{S}_2) = (\langle\rangle, s_2 :: S_2)$ :
   return $Q$;
case $(\mathcal{S}_1, \mathcal{S}_2) = (s_1 :: S_1, s_2 :: S_2)$
   $(t, v_1, v_2)$ = StatusIntersection($s_1, s_2$);
   if $(t = \mathsf{T})$
      forall($e \in s_1, s_2$)
         if (onEdge($v, e$) = T)
            $Q$ = InsertEvent($Q, (v_1, \{e\}, \{e\})$);
   return $Q$;
```

Fig. 44. Map Overlay Algorithm: Event Queue and Intersection Detection

information is stored for each segment in the status structure, it is simply retrieved by a look-up and stored in f_{above} and f_{below}.

Event Queue

The event queue contains the same elements as the line intersection algorithm: Q contains triples (v, E_1, E_2), where v denotes the event point position, and E_1 contains edges that start (lexically smaller endpoint is v) and the edges E_2 that end (lexically greater endpoint is v) there. The functions that modify the event structure are given in Figure 44. Initially, InitialiseEvents inserts all vertices of the input maps in this event queue, whereas intersections are added in the course of the plane sweep: Each time an edge is inserted or removed in the status structure S, it is checked whether the new neighbor edges intersect (CheckIntersections). If this is the case, a new intersection event is inserted in the event queue Q, encoded as a start and end event of all edges that run through the vertex (as in the network overlay).

Status Structure

The status structure is very similar to the one used in the network overlay. In particular, it uses the same order function. In addition to the information that is stored in the network overlay, more

```
function StatusInsertSet($S, \mathcal{E}$)
    forall ($e \in \mathcal{E}$)
        $S$ = StatusInsert($S, e$);
    return $S$;

function StatusInsert($\mathcal{S}, e$)
    switch
    case $\mathcal{S} = \langle\rangle$ :
        return $\langle\langle e\rangle\rangle$;
    case $\mathcal{S} = s :: S$ :
        switch
        case StatusCompare($s, e, =$)T :
            return $s$ :: StatusInsert($S, e$);
        case StatusCompare($s, e, =$)U :
            return StatusCollinear($s, e$) :: $S$;
        case StatusCompare($s, e, =$)F :
            return $\langle e\rangle :: s :: S$;

function StatusCollinear($s :: S, e$)
    if (RightEnd($e$) $\prec$ RightEnd($s$) = T)
        return $s$::StatusCollinear($S, e$);
    else
        return $s::S$;
```

```
function StatusCompare($p, e_1, e_2$)
    $y_1$ = yValueAt($e_1$, X($p$));
    $y_2$ = yValueAt($e_2$, X($p$));
    if ($y_1$ = U) and $y_2$ = U)
        return U;
    if ($y_1$ = U) and $y_2 \neq$ U))
        return ($y_2 <$ Y($p$) $\Rightarrow$ T| F);
    if ($y_1 \neq$ U) and $y_2$ = U))
        return ($y_2 <$ Y($p$) $\Rightarrow$ F| T);
    if ($y_1 \neq y_2$)
        return ($y_1 < y_2 \Rightarrow$ T| F);
    $\delta_1$ = slope($e_1$);
    $\delta_2$ = slope($e_2$);
    if ($y_1 <$ Y($p$))
        return ($\delta_2 < \delta_1 \Rightarrow$ T| F);
    else
        return ($\delta_1 < \delta_2 \Rightarrow$ T| F);
    return U;
```

Fig. 45. Map Overlay Algorithm: Status Structure

attributes are stored for the map overlay: $curRaw(s)$ denotes the sum of all labels from all input maps for the area above the segment s. The set of actual output labels $curLabels(s)$ is derived from this set according to the label overlay operation computeLabels. The field $curFace(s)$ is used to store the output face above the segment, which is used to determine the variables f_{below} and f_{above}. The function calls $curRaw(\perp)$, $curLabels(\perp)$ and $curFace(s)$ return the labels and the face for the unbounded face, respectively.

The edge handling, which will be explained in the following, is completely different to previous approaches [36] (see Figure 46):

For each segment s that ends in or passes through the event point (HandleEdgesEnd), a new edge e is created and it is linked to the vertex. With the help of $curFace(s)$ it can be linked to the adjoining output faces by setting the fields $leftFace(e)$ and $rightFace(e)$. The first segment is handled separately, since it does not need to set a $outerComp(f)$ information. This linking of the face to its outer boundary must be done once (at least), when its right end is reached in the plane sweep[1]. The counter n_{end} keeps track of the number of created edges on the left-hand side of the event point.

[1] As this information may be also set for the unbounded face of the map (if the lowermost segment is redundant with respect to the labelling), the main function of the map overlay resets this information after the plane sweep to $outerComp(unbdFace(M)) = \perp$. Previous checks that prevent this setting would be more expensive.

```
function HandleEdgesEnd()
    $s = \mathsf{SegLowmst}(S)$
    if $(\mathit{curFace}(\mathsf{prev}(s)) = \mathit{curFace}(s))$
        $e_{\text{last}} = \mathsf{newEdge}()()$
        $\mathit{edges}(M) = \mathit{edges}(M) \cup \{e\}$
        $\mathit{src}(e) = \mathit{curStart}(s)$
        $\mathit{dest}(e) = v$
        $\mathit{leftFace}(e) = \mathit{curFace}(s)$
        $\mathit{rightFace}(e) = \mathit{curFace}(\mathsf{prev}(s))$
        $n_{\text{end}} = n_{\text{end}} + 1$
    for$(s = \mathsf{next}(\mathsf{SegLowmst}(S)) \ldots \mathsf{SegUpmst}(S))$
        if $(\mathit{curFace}(\mathsf{prev}(s)) = \mathit{curFace}(s))$
            $e_{\text{last}} = \mathsf{newEdge}()()$
            $\mathit{edges}(M) = \mathit{edges}(M) \cup \{e\}$
            $\mathit{src}(e) = \mathit{curStart}(s)$
            $\mathit{dest}(e) = v$
            $\mathit{leftFace}(e) = \mathit{curFace}(s)$
            $\mathit{rightFace}(e) = \mathit{curFace}(\mathsf{prev}(s))$
            $n_{\text{end}} = n_{\text{end}} + 1$
            $f = \mathit{curFace}(\mathsf{prev}(s))$
            $\mathit{outerComp}(f) = e_{\text{last}}$
```

```
function HandleEdgesBegin()
    $s = \mathsf{SegLowmst}(S)$
    $s_{\text{last}} = \mathsf{SegLowmst}(S)$
    $f_{\text{last}} = f_{\text{below}}$
    $\mathit{curStart}(s) = v_{\text{last}}$
    $\mathit{curRaw}(s) = \mathit{curRaw}(\mathsf{prev}(s)) + \mathsf{Labels(s)}$
    $\mathit{labels}(s) = \mathsf{computeLabels}(\mathit{curRaw}(s))$
    if $(\mathit{curLabels}(s) \neq \mathit{curLabels}(\mathsf{prev}(s)))$
        $n_{\text{begin}} = n_{\text{begin}} + 1$
    for$(s = \mathsf{next}(\mathsf{SegLowmst}(S)) \ldots \mathsf{SegUpmst}(S))$
        $\mathit{curStart}(s) = v_{\text{last}}$
        $\mathit{curRaw}(s) = \mathit{curRaw}(\mathsf{prev}(s)) + \mathsf{Labels(s)}$
        $\mathit{curLabels}(s) = \mathsf{computeLabels}(\mathit{curRaw}(s))$
        if $(\mathit{curLabels}(s) \neq \mathit{curLabels}(()\mathsf{prev}(s)))$
            $n_{\text{begin}} = n_{\text{begin}} + 1$
            if $(\mathit{labels}(f_{\text{last}}) \neq \mathit{curLabels}(\mathsf{prev}(s)))$
                $f_{\text{last}} = \mathsf{newFace}()$
                $\mathit{faces}(M) = \mathit{faces}(M) \cup \{f_{\text{last}}\}$
                $\mathit{labels}(f) = \mathit{curLabels}(\mathsf{prev}(s))$
                for $(t = s_{\text{last}} \ldots \mathsf{prev}(s))$
                    $\mathit{curFace}(t) = f_{\text{last}}$
                $s_{\text{last}} = s$
    for $(t = s_{\text{last}} \ldots \mathsf{SegUpmst}(S))$
        $\mathit{curFace}(t) = f_{\text{above}}$
```

Fig. 46. Map Overlay Algorithm: Handling Edges

If at least two edges start at the current event point (HandleEdgesBegin), a new face is created in between. Its labels are computed by taking the labels of the adjacent face and adding the differences caused by the edge in-between. The multiset $\mathit{curRaw}(s)$ stores the sum of all labels, whereas $\mathit{curLabels}(s)$ gives the actual labeling that is defined by the desired operation (see Section 6.3.3). A tricky detail is the setting of the face information $\mathit{curFace}(s)$ of each segment. As the current face is not known when passing a segment of the status structure, the setting is done blockwise after the creation of a new face (or at the end of the handling function) beginning at the last processed segment s_{last}. Analogously to the left side, the counter n_{begin} keeps track of the number of created edges on the right-hand side of the event point.

The handling of the starting and ending edges almost accomplishes all tasks of the map overlay algorithm. In addition, one event must be handled separately (HandleMerge): It may be the case that there may be no edges starting at the current event point (indicated by $n_{\text{begin}} = 0$). At these points, the face above and below the event point must be the same, since there is no division between them on the right-hand side of the event point. Two cases must be distinguished for the consecutive merge: First, the faces may be independently created by the algorithm. In this case, the two branches are merged by deleting the face f_{above} and all references to it. This can be done very efficiently if indirect references are used for the face pointers. In the second case, the current event point marks a right end of an inner component, to which the face f can be linked at this point by setting the $\mathit{innerComp}(f)$ field.

```
function HandleMerge()
  if ($n_{\text{end}} > 0$ and $n_{\text{begin}} = 0$)
    if ($f_{\text{below}} = f_{\text{above}}$)
      $innerComp(f_{\text{below}}) :=$
          $innerComp(f_{\text{below}}) \cup \{e_{\text{last}}\}$
    else
      $innerComp(f_{\text{below}}) :=$
          $innerComp(f_{\text{below}}) \cup innerComp(f_{\text{above}})$
      $faces(M) := faces(M) \setminus \{f_{\text{above}}\}$
      /* relink all $f_{\text{above}}$ pointers to $f_{\text{below}}$ */
```

Fig. 47. Overlay Algorithm: Face Merging

6.3.3 Adaption

The map overlay algorithm is a powerful tool that can be used to solve a lot of other problems. For example, the computation of Boolean operations on polygons can be reduced to it: First, each polygon is transformed to an input map with a different labelling. Then, the overlay is computed with the desired Boolean function over the labels of the input maps.

Using the same labels in several input maps and adding them by the labelling overlay function makes it possible to implement an operation that compute areas that are covered by a given number of input polygons. The computation of the output labels in computeLabels is completely separated from the rest of the algorithm; only its interface is fixed.

Hence, the map overlay in the polygon-processing library can be adapted to particular needs for the computation of the actual labels from the input labels.

6.3.4 Verification

The network overlay algorithm was verified within the theorem prover. A similar proof can also be constructed for the map overlay, since both algorithms share a lot of common concepts. For the construction of the vertices and the edges, the verification of the network overlay can be even taken as a basis. However, the map overlay algorithm is not formalised in the HOL system, since the mathematical basis is not available in the theories of the theorem prover.

Certainly, it would be interesting to integrate the map overlay into the system, and this is theoretically no problem, but the effort to build the underlyings theories is too high for a simple thesis: A comprehensive theory of planar graphs would be one of a number of prerequisite, which are currently not available.

Hence, the crucial part of map overlay algorithm is only checked here: the labelling of the faces. To see that the faces are correctly labelled, a closer look at the handling of the starting edges is needed. A face should be labelled by the sum of all labels that are associated with the edges, a ray from the face to the exterior crosses — similar to the parity algorithm of Section 4.7. Hence, when the ray that leads from a point inside the face to the bottom is considered, it is evident that the map overlay algorithm computes the right set of labels. The computation of the actual labels can be easily checked, as the operation computeLabels just corresponds to the specification of the overlay operation.

6.4 Conservative Rounding

While degenerate cases are a major pitfall for the design of computational geometry algorithms, even more problems arise for their implementation: Geometric primitives must be efficiently computed with limited-precision arithmetic, and limited memory may lead to the situation where the result of an algorithm cannot be exactly represented. This section presents particular solutions to problems that appear for map overlay algorithms used in a safety-critical environments.

Without further modifications to the map overlay algorithm, arbitrary-precision arithmetic would be required for its implementation, which is too complex for non-trivial examples: Consider the sequence of overlay operations $O_1, O_2, \ldots, O_n$, where the results of O_i is a map M_i that is taken to build the inputs of the following step O_{i+1}. Assume that the positions of the vertices $\mathcal{V}_i$ in step i are represented by an integer of n bits. Then, the vertices $\mathcal{V}_{i+1}$ have either been in one of the sets of vertices before or a new vertex is created at the position of an intersection of two edges, which requires about $4n$ bits ($2n$ for both the numerator and the denominator) for its representation. As a consequence, the size of the results grows exponentially (about $n \cdot 4^i$ bits are needed in step i). This exponential blow-up fills the available main memory after a few steps as experienced with an implementation based on the GNU Multiprecision library [54].

A better approach is to limit the precision of the numbers after each step in a way that complies with the application requirements. If the position of all vertices are rounded to values of the initial precision and all geometric predicates of a step can be computed correctly for this precision, a solution for the problem is found. This process can be visualised by some kind of grid snapping, where the cross-over points represent all positions that are representable in the initial precision.

In contrast to general-purpose solutions, the safety-critical context requires that approximations are not made arbitrarily. The loss of information that is caused by the limited precision may be fatal, since it may lead to unwanted reactions of the system. For instance, consider a situation-analysis system: Faces are used to describe inaccessible regions in the environment. If a path were chosen that leads through the exact faces but not through the rounded ones, the rounding would cause a potential crash. Hence, traditional snap-rounding or similar approaches [56, 65, 68, 75, 99] cannot be used for the approximation in safety-critical applications. They are merely an inspiration for the development of the conservative rounding functions.

As all positions must be rounded in a way that the labelled areas of the rounded map cover at least the original area, the algorithm cannot simply choose one of the neighbor grid points as shown in Figure 48 (since for all neighbors a part of the original area is cut off). Instead, some vertex at a further distance must be chosen (case (b) of Figure 48). Thus, the problem can be defined as follows:

Definition 11 (Conservative Grid Snapping). *For a given map M, the conservative grid snapping problem consists of finding an approximation M' with the following properties:*

- *The set of vertices $vertices(M)$ only contains vertices on the grid.*
- *For each label ℓ, all faces of $faces(M')$ that are labelled with ℓ cover at least the area that has been labelled by the ℓ faces of $faces(M)$.*

Needless to say, grid snapping involves some error that should be kept as small as possible by the rounding algorithm. Moreover, it should be clear that finding the optimal solution is very complex. Hence, for the embedded domain, only a fast and simple heuristic is acceptable.

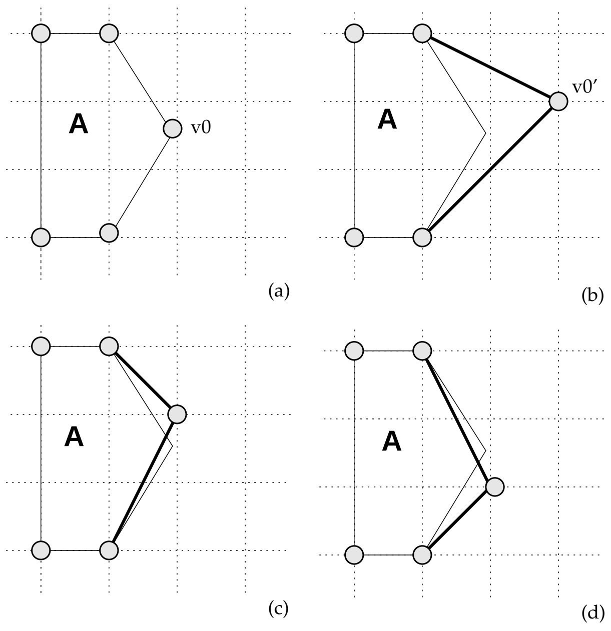

Fig. 48. Rounding a Vertex

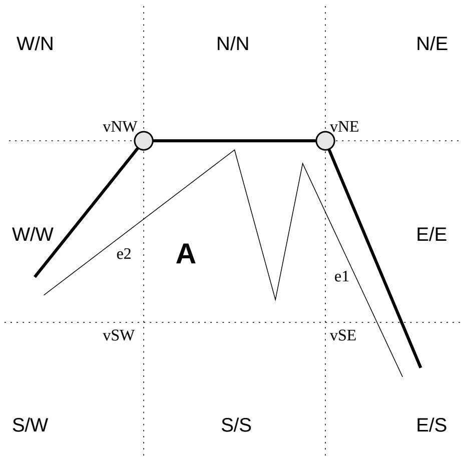

Fig. 49. Entry/Exit Codes Example

To solve this problem, a vertex that is not on the grid is approximated by several vertices that lie on the grid around the original vertex. Despite this simple idea, the actual algorithm is a bit more complicated, since all inputs must be considered. In the following, the considerations are limited (w.l.o.g.) to maps that have only one label: Multi-labelled maps can be handled as well by rounding the vertices for each label separately.

The algorithm given in Figure 50 does not round all vertices separately. For each rounding step, a chain of vertices that are within the same grid box are processed simultaneously. The algorithm first identifies the vertices that precede and follow the chain that is rounded (with the face to be conservatively rounded to the left). For them, an outcode similar to the one of the Cohen-Sutherland-algorithm (see Section 4.5.2) is computed. Based on this outcode, an entry code and an exit code are determined for both of the two adjoining vertices. Depending on these codes, a

```
function RoundVertices($w, e_1, e_2$)
    $v_1 := src(e_1)$
    $v_2 := dest(e_1)$
    $v_3 := src(e_2)$
    $v_4 := dest(e_2)$
    $(c_0, c_1) :=$ EntryExit($w, v_1, v_4$)
    if ($c_0 = c_1$ and rturn($v_1, v_2, v_3$)$\vee$
            collinear($v_1, v_2, v_3$) $\wedge$ rturn($v_1, v_3, v_4$)))
        return $\langle v_1, v_4 \rangle$
    else
        switch ($c_0$)
        case NORTH :
            if ($c_1 =$ WEST)
                return $\langle v_1, \mathsf{NW}(w), v_4 \rangle$
            else if ($c_1 =$ SOUTH)
                return $\langle v_1, \mathsf{NW}(w), \mathsf{SW}(w), v_4 \rangle$
            else if ($c_1 =$ EAST)
                return $\langle v_1, \mathsf{NW}(w), \mathsf{SW}(w), \mathsf{SE}(w), v_4 \rangle$
            else
                return $\langle v_1, \mathsf{NW}(w), \mathsf{SW}(w), \mathsf{SE}(w), \mathsf{NE}(w), v_4 \rangle$
        case EAST : ...
        case SOUTH : .../* other cases analogously */
        case WEST : ...
```

Fig. 50. Vertex Rounding Algorithm

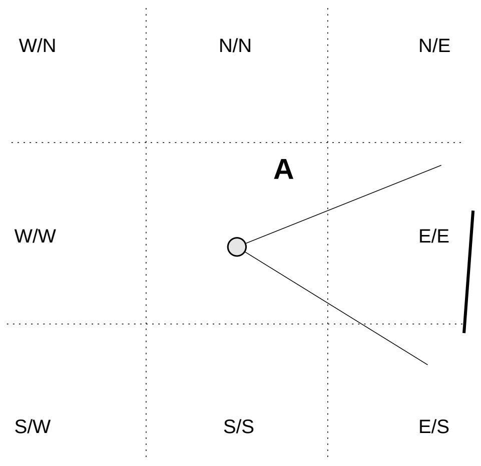

Fig. 51. Entry/Exit Codes Example

sequence of corner points of the grid box is taken to approximate the map, where the basic idea is to walk around the grid box in counter-clockwise direction. Figure 49 shows the mapping between the corner points and the entry and exit codes.

However, there are two special cases that cannot be handled by this approach and therefore require a separate consideration: The first case occurs when there are no preceding and following vertices, i. e. the contour is completely inside a grid box. In this case, this grid box is the approximation. In the second case, the preceding and following vertices have the same code, and the part describes a concave part of the face (see Figure 51). Then, no local approximation is possible. A very simple solution is to just remove all intermediate vertices and to insert an edge that directly connects the preceding vertex to the following vertex.

Although tests have shown that this leads to good results (see the experimental results in Section 7.4), the error caused by this rounding might be unacceptable. Therefore, the following alternative can be integrated: Assume that the edges both cross the right border of the grid box (w.l.o.g.), then the rounding point is chosen as follows: First, slopes of the incoming and outgoing edges s_{in} and s_{out} are determined and the x-value where both edges have at least a distance of the height of the grid box $x_{\mathsf{NE}} - x_{\mathsf{SW}}$ is calculated. Assuming the worst case, i. e. the edges almost touch at the right border of the grid box, the vertex v can be approximated by $v = (x', y')$ by

$$\begin{aligned} x' &= \lceil x_{\mathsf{NE}} + \tfrac{x_{\mathsf{NE}} - x_{\mathsf{SW}}}{s_{\text{in}} - s_{\text{out}}} \rceil \\ y' &= \lceil \mathsf{yValueAt}(e_{\text{in}}, x') \rceil \end{aligned}$$

With the exception of the last optimisation, the algorithm does not rely on a specific grid. In particular, the grid does not need to be equidistant. Hence, it can be used for integers, and also for rational and floating point numbers with limited precision. In particular, it can be used for floating points with a quarter of the available precision for the input vertices positions and round the results, which need quadruple precision. Another possibility is to use a double or quadruple-precision arithmetic as already described in detail by Knuth [86].

Unfortunately, vertices cannot be simply moved or replaced in a map as done in the algorithm above, since the planarity property of the map may be lost by overlapping faces. Therefore, after the rounding step, another overlay of the rounded map must be computed.

6.5 Map Simplification

Embedded systems generally possess very limited resources with respect to computation power and memory. As they often have to meet real-time requirements, the simplification of maps is an integral part of the design of an overlay algorithm for embedded devices. The simplification operation aims at the elimination of vertices, edges or faces to reduce the overall computation complexity. Similar to the rounding of vertices, the simplification of the maps [37] must be conservative, i. e. labelled areas must not become smaller. Clearly, the simplified map should resemble the old map while using less vertices and edges.

Generally, any bounded labelled area of a map can be approximated with three vertices forming a triangle that covers the colored area. A very simple approximation requiring four vertices is the bounding box. Moreover, if the map has a lot of jagged areas, the convex hull of them is good simplification. All these approximations can be efficiently calculated, but they are too coarse for practical use.

Therefore, a local approach is used again to simplify maps iteratively. It is based on the following two operations (see Figure 52):

(a) Remove a vertex at a concave position of a labelled area (including redundant vertices that connect two collinear edges). The error of this operation is

$$\varepsilon = \frac{1}{2} e_1 \times e_2 \tag{8}$$

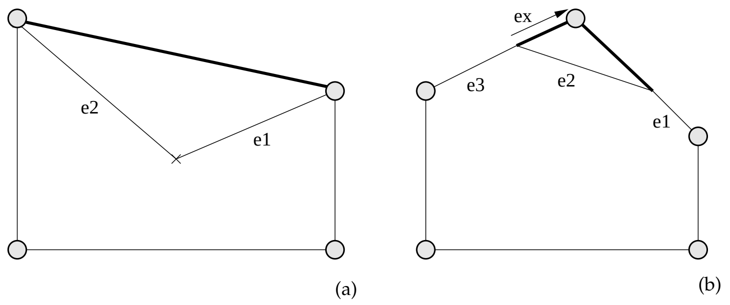

Fig. 52. Basic Simplification Operations

(b) Replace two vertices by a single one at a convex position of a colored area. The position of the new vertex is the intersection of the lines through the two adjoining edges e_1 and e_3, which can be computed as follows: Since $\lambda e_1 - \mu e_3 = e_2$ (for $\lambda, \mu \in \mathbb{Q}$ with $\lambda > 0, \mu < 0$), Cramer's rule can be applied to compute λ, μ and e_x:

$$\lambda = \tfrac{e_2 \times e_3}{e_1 \times e_3} \; \mu = -\tfrac{e_1 \times e_2}{e_1 \times e_3} \; e_x = \tfrac{e_1 \times e_2}{e_1 \times e_3} \cdot e_3$$

Hence, the error of the simplification step is

$$\varepsilon = \frac{1}{2}\left(\frac{e_1 \times e_2}{e_1 \times e_3} \cdot e_3 \times e_2\right)$$

The simplification of a map can be either performed with respect to an upper bound of the error or w.r.t. to a projected number of vertices. Whereas the first strategy is good for a general speed-up of the computation, the second approach is chosen here. Since embedded systems must usually hold tight deadlines, the worst case execution time of the map overlay, which is primarily affected by the input complexity, should be restricted. Moreover, an unlimited number of vertices may cause that the embedded system runs out of memory. Needless to say, the results may become less useful after the simplification. However, for hard real-time systems, violating the deadline is worse than any approximated result. In order that other components of the system can judge how trustworthy the results of a map overlay are, the simplification procedure additionally returns the computed error.

However, even if the above operations for the simplification are considered, finding the optimal map with a minimal number of vertices that covers a given map is still a complex problem. Since the removal of a vertex changes the errors that are made by the removal of other vertices, the problem apparently falls into the class NP. Hence, a simple heuristic is used again, which follows a greedy approach: Among all vertices, the one with the least error is selected. This is repeated until the desired number of vertices is reached. With this approximation process, the best solution for a given system and a situation is calculated.

Unfortunately, the simplification suffers from the same problem as the rounding. After each approximation step, the map may no longer be planar: The overlay must be computed again after the simplification. Even worse, new vertices may not be on the grid, so that the rounding must be done as well. This leads to the natural sequence of operations: First, simplify the polygons, then align them to the grid and finally planarise them with an additional map overlay.

Chapter 7

Situation-Analysis Systems

To demonstrate the polygon-processing library presented in the previous chapters, it is applied to an embedded situation-analysis system for autonomous mobile robots. With its help, the parameters of the algorithms can be adjusted, and the performance of the algorithms can be evaluated. The adaptation step is specific for the application, since the overall performance highly depends on the interaction between system parameters and environment conditions.

A situation-analysis system generally monitors the environment and tracks objects in its neighbourhood. Speed and the direction of other objects are detected by sensors and are given as inputs to the system. Depending on their movement, regions are classified according to some application logic, and some decisions are taken on this base. For instance, collision detection and motion planning systems [87, 90, 120] as applied in robotics fall into this class of applications.

In Section 7.1, three-valued regions are proposed as the basis for situation-analysis systems. Section 7.2 presents the propagation, the fundamental operation for area calculations in these systems. Section 7.3 illustrates various algorithms that can be implemented with the presented primitives. Finally, Section 7.4 gives an impression about the performance of the developed polygon-processing library and compares it to other implementations.

7.1 Three-Valued Regions

Motion planning and collision avoidance algorithms involve various objects in an observed area. Often, there is a designated object whose viewpoint is taken and for which a path should be planned. This object will be denoted by the term *ego-object* in the following. Moreover, there are other objects in its environment that may cause collisions and therefore restrict its mobility. They will be called *obstacles* in the following.

The observed area is modelled as an Euclidean plane, and objects can be represented by polygons there. However, more important than the actual objects are the estimated regions where the objects are expected. As this involves some amount of uncertainty, these regions are described by three-values characteristic functions, i. e. the plane is divided in a number of subdivisions, where each subdivision is labelled with on of the tree truth values T, U or F.

There are two reasons for the three-valued modelling of areas: First, the sensoric systems may not be very precise, which leads estimations of the position of the obstacles. Second, if situations in the future are considered, the behaviour of the obstacles must be guessed. The three truth values are interpreted differently for the ego-object and the obstacles:

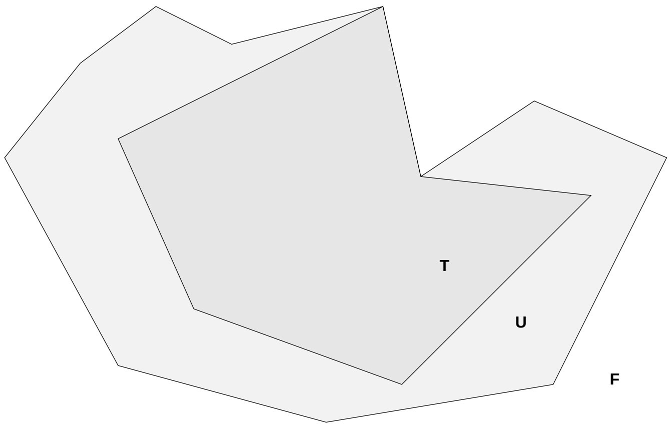

Fig. 53. Must-Region (T) and Can-Region (U)

For the ego-object, regions labelled with T are unconditionally reachable, i. e. independent from the movement of the obstacles, there is a trajectory to each point in areas marked with T. For points in the conditionally reachable areas, which are marked by U, there are only trajectories that may be interrupted by obstacles, depending on their movement. Finally, areas labelled with F are unreachable, whatever movements the obstacles make.

For the obstacles, regions labelled with T denote positions where the object *must* be, while U denotes positions where it *can* be. F is assigned to areas where the object *cannot* be.

In the following, the regions for both groups of objects will be called *can-regions* (where the characteristic function results to U or T) and *must-regions* (where the characteristic funtion results to T), which is illustrated in Figure 53. Obviously, the can-region always contains the must-region by construction. In the case that the position of an object is absolutely sure, both regions are equal and denote the actual position. In particular, this is the case for immobile obstacles. Usually, there is a difference between both areas, which has shown to be very moderate for realistic input parameters. High speeds and limited steering angles of the objects make the estimated position relatively precise.

Since must-regions and can-regions are approximations, the conservative rounding and simplification scheme presented in Sections 6.4 and 6.5 can be applied for them. What a conservative rounding means for the regions depends on the particular application. As a general rule of thumb, can-regions are over-estimated and must regions are usually underestimated.

7.2 Propagation

The algorithms presented in the next section examine the state of the environment at discrete instances. Computations and estimations are performed in steps. This requires that the moving of the objects towards each other is not too fast relative to their distance and size. Otherwise, intersections of them may be missed, which is sometimes referred to as temporal aliasing, following to aliasing in digital signal processing, which is a result of undersampling.

The areas defined in the previous paragraphs change over time. In general, the longer the algorithm looks into the future, the uncertainty increases, and the difference between can-regions and must-regions increases. The modification of an area from one step to another is computed by a process called propagation. It takes the vertices of the regions and translates them according

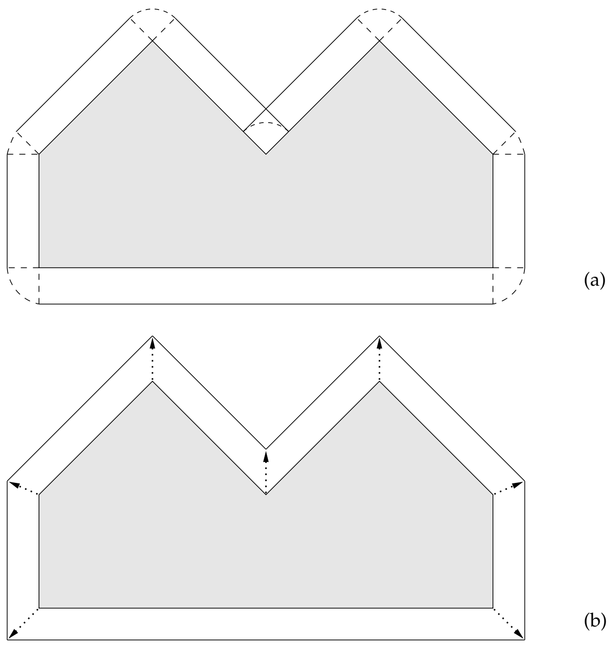

Fig. 54. Propagation of Can-Regions

to some estimations. Naturally, can-regions are enlarged, and must-regions are made smaller by propagation. In the following, the propagation of the can-regions is described; must-regions are processed analogously.

A propagation step can be described by movements of the edges of a region. They are moved orthogonally in the length of the maximal distance that the object can cover (see Figure 54 (a)). For a precise solution, the gaps between the moved edges must be filled with arcs. Within the linear framework of the polygon-processing library, the arc is approximated with a number of vertices[1]. In the simplest form, one vertex is taken at the intersection of the extended edges (see Figure 54 (b)), although this causes a big error for acute angles. Hence, algorithms that compute the propagation can determine for each vertex of the old map a number of new vertices.

However, it is not possible just to move the vertices, since self-intersections may occur as already seen when simplifying polygons. Without an overlay operation, the resulting map would not be planar any more. Therefore, the propagated area must be computed by unifying the old region and the new parts, which are given by the propagation vectors and the old and new edges. These quadrangles form a ring around the old area as shown in Figure 54 (b).

Thus, the propagation can be basically implemented as follows: The algorithm iterates over the vertices of the map. For each of them, it determines the propagation vector and adds it to the old position. The old and the new positions of two consecutive vertices are taken to form a propagation triangle that is added to the old area. One problem causes some more work: As the propagation is computed in steps and not continuously, the propagation vectors may intersect as shown in Figure 55. For this propagation quadrangle, the new part must be planarised by computing the intersection and adding both triangles. Another possibility is to add a quadrangle covering the original one by swapping the outer points.

[1] Note that the approximation must be always conservative.

This naive implementation is quite slow. The reason for this is that the map overlay algorithm, which is used for the area addition, has relatively high fixed costs, which do not depend on the input size. Thus, the number of overlay operations should be reduced to a minimum. This can be simply done by adding all quadrangles at once.

Another possibility to reduce the execution time is to reduce the number of inputs. Considering the propagation operation, there are a lot of input edges that can be discarded a priori:

- The propagation quadrangles always have one edge in common with the old region. Since the direction of the respective two edges is in opposite direction to each other, they can be eliminated in advance. The overlay algorithm would do the same thing, but this approach decreases the number of event points as well as the number of edges. Otherwise, they would have to be inserted and removed, which involves to find a position in the status structure, etc.
- Two adjoining propagation quadrangles also have an edge in common. However, its must be checked whether both of them are not self-intersecting. For this case, the optimisation cannot be done.

The propagation function can be implemented with the algorithm sketched in Figure 56. It creates a set of edges that are given to the map overlay algorithm as an input. The call of PropEdge(v_1, v_2) creates an edge between the vertices v_1 and v_2, where the face of its left is set to the inner face of the propagated area. The swapping in the case of the intersection of the propagation edges is done in the else-part by constructing a sequence of edges with the same start and end vertex as for the simple case.

7.3 Situation-Analysis Algorithms

In the previous section, the propagation operation was described. If several of these operations are concatenated, situation-analysis systems can be derived. They simulate the movements of the objects in the environment with respect to the possibilities of the ego-object. Depending on the actual properties of the objects, the regions can be computed for a reasonable time span in the future and decisions can be derived from this information.

All these systems require velocity and steering information of the objects to compute the development of the regions. The descriptions abstract from this, as this is not the scope of the work. They just provide the infrastructure for real systems, which can provide the actual parameters.

Moreover, the following algorithms are not constructive. They only check whether paths exist. The actual construction would require to store trajectories with various positions of the computed areas. As this is not the scope of the work, it is abstracted from this.

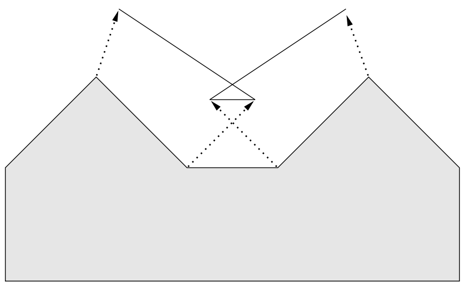

Fig. 55. Self-Intersecting Propagation Quadrangles

```
function PropagateBeginContour()
  $\mathrm{prevOut} = \mathrm{prevIn} + p_{n-1}$

function PropagateVertex()
  $\mathrm{curOut} = \mathrm{curIn} + p_0;$
  if ($\neg$doSegmentIntersect(mkLine(prevIn, prevOut), mkLine(curIn, curOut)))
    PropEdge(prevOut, curOut);
  else
    PropEdge(prevOut, prevIn)
    PropEdge(prevIn, curOut)
    PropEdge(curOut, prevOut)
    PropEdge(prevOut, curIn)
    PropEdge(curIn, curOut)
  $\mathrm{prevIn} = \mathrm{curIn};$
  $\mathrm{prevOut} = \mathrm{curOut};$
  for $(i = 0 \ldots n-1)$
    $\mathrm{curOut} = \mathrm{curIn} + p_i;$
    PropEdge(prevOut, curOut)
    $\mathrm{prevOut} = \mathrm{curOut};$
```

Fig. 56. Propagation Algorithm

7.3.1 Motion Planning and Collision Avoidance

The collision avoidance problem consists of determining a path from a starting position to an end position respecting the obstacles in the environment. Classically, only static objects are considered [93]. With the help of the propagation, movement can be considered as follows: The can-region and the must-region of the ego-object in the next step is computed. Analogously, the regions for all obstacles are determined. These pieces of information are then combined to a new three-valued description of the ego-object's position (see Figure 57):

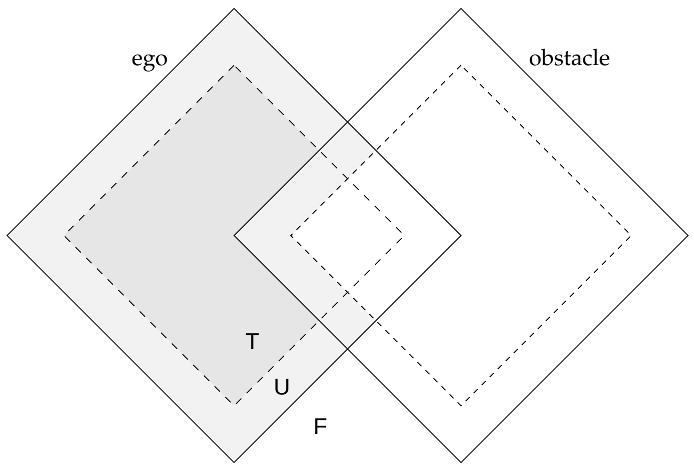

Fig. 57. Ego-Region Constraint

f	F	U	T
F	F	F	F
U	U	U	F
T	T	U	F

Fig. 58. Ego-Object Constraint Operation

- Regions that lie in the reachable area of the ego object and that are not in the can-region of any obstacle are surely reachable.
- Regions that lie in the reachable area of the ego-object and that are in the can-regions of the obstacles are potentially reachable.
- Regions that lie in the reachable area of the ego-object and that are in the must-regions of the obstacles are not reachable without a collision.
- All other regions are not reachable at all.

After each step, the overlay of the maps is computed to get the new region of the ego-object. This is done according to the function $f = x \ddot{\wedge} \ddot{\neg} y$ shown in the table of Figure 58. After the overlay, the regions are rounded according to the following scheme: can-regions of the obstacles are over-estimated, must-regions also, regions of the ego-object are under-estimated.

The process is reiterated until the endpoint is in the reachable area of the ego-object. If this is the case, there is a path from the start to the end that is statically computable to reach the endpoint without any collision. If for each region, a path has been recorded in the course of the computation, this can be simply taken. If, after the observation interval, the endpoint is in the potentially reachable region, there may be a route within the observable time frame to the endpoint, but it depends on the movement of the obstacles. If the point is not in the regions, it may be reachable outside the considered timeframe. The existence of a path can be excluded, if there is no potentially reachable region left at a point in the future.

The collision avoidance problem is completely identical to the motion planning problem. The only distinction is that no target is fixed in advance. Instead, the ego-object should be controlled in a way that no collision occurs for a given time interval.

7.3.2 Collision Expectation

The collision expectation is a similar problem. The ego-object is controlled by another system. The situation-analysis system just has the task to identify situations where a collision with other objects is unavoidable. When such a collision is expected, safety-measures can be taken — for example, shutting down systems that are in danger in the case of an collision.

For the collision expectation problem, the computations are made like in the motion planning algorithm, with one exception: The area of the ego-object is over-estimated, while the area of the obstacles are under-estimated. Hence, a collision is only detected if the system can be absolutely sure that it is unavoidable. Furthermore, the must-region of the ego-object is not interesting as well as the can-regions of the obstacles. Hence, only the can-region of the ego-object and the must-region of the obstacles are computed for a better performance.

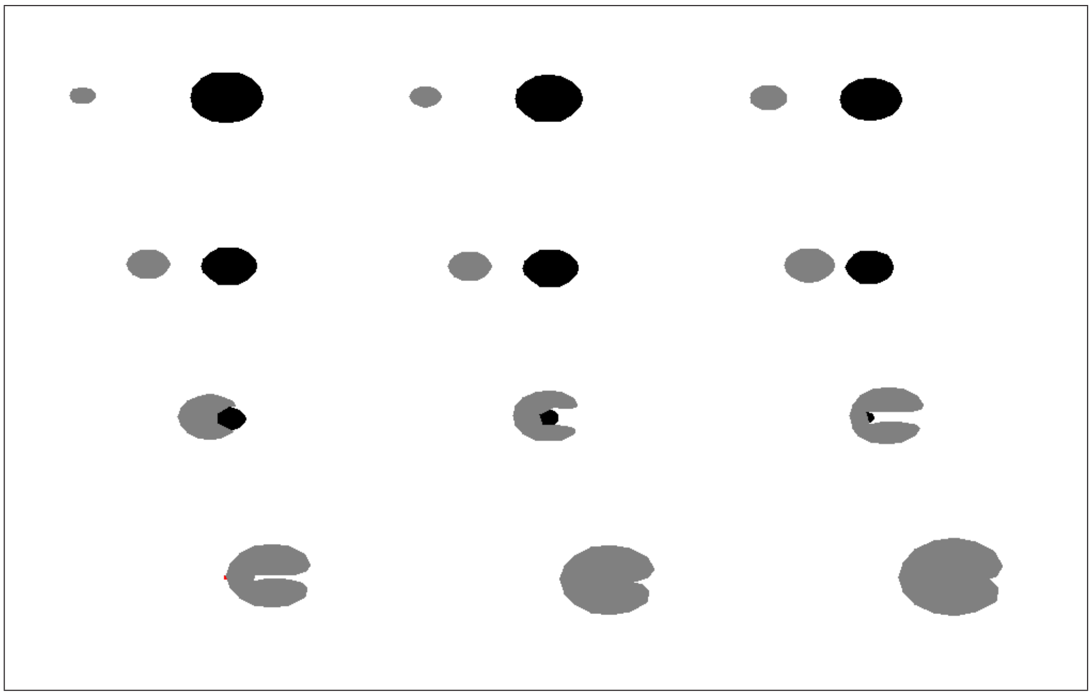

Fig. 59. First Scenario: A Single Mobile Obstacle

7.4 Experimental Results

A prototype of the collision expectation algorithm was implemented in the program language C. For the propagation and the constraint operation it uses the developed polygon-processing library, in particular its map overlay algorithm. The collision expectation system was given inputs of various situations and its outputs as well as its run-time was recorded.

Figure 59 shows a scenario with one moving obstacle (black). The ego object (grey) start at the left-hand side of the plane and moves straight to the right. The obstacle, albeit mobile, remains at its position. In the course of the scenario, the can-region of the ego objects moves to the right while steadily getting wider and wider. In contrast to this, the must-region of the obstacles gets smaller to the same extent. When the can-region of the ego object hits the must-region of the obstacles, it gets split up. Behind the obstacles, both parts finally reunite.

Figure 60 shows a much more complex scenario with two moving and 17 immobile obstacles. The ego object again starts at the left-hand end of the plane and moves straight to the right. The two moving obstacles approach from the right. Their must-regions soon get smaller and smaller and finally, they disappear (between frame 5 and 6). The can-region of the ego-object gets bigger and bigger and flows around the fixed obstacles. On the right side of these objects, there is at first a shadow.

For a performance comparison, these two and other scenarios were executed on a Pentium IV (2.8 GHz) with hyperthreading disabled. All programs were compiled under an Ubuntu Linux 6.6 LTS with the GNU C Compiler in version 4.1 at optimisation level 3.

For all scenarios, the speed and the direction of the ego-object and the obstacles were arbitrarily chosen, and the analyses were based on a prediction of 300 steps. Table 61 shows the execution time for 1000 consecutive runs with different parameters. The second row corresponds the scenario of Figure 59, the fourth row corresponds to the one of Figure 60.

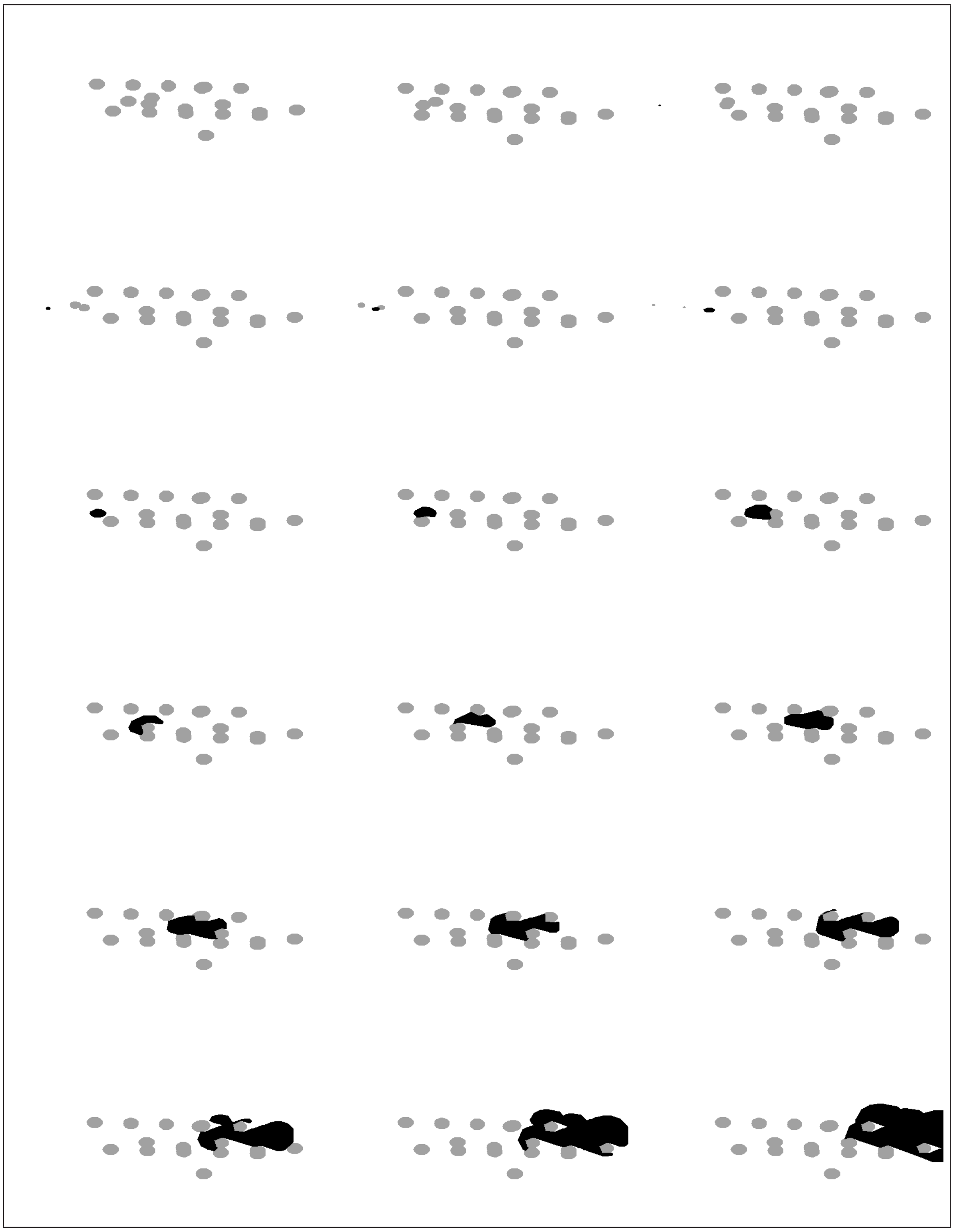

Fig. 60. Second Scenario: 2 Mobile and 17 Immobile Obstacles

obstacles	GPC Library	(EPP w/o simplification)	EPP (with simplification)
0	120 sec	224 sec	293 sec
1 mobile	132 sec	311 sec	223 sec
7 immobile + 2 mobile	1210 sec	1345 sec	980 sec
17 immobile + 2 mobile	6757 sec	3987 sec	2565 sec

Fig. 61. Measured Run-Time for Collision Avoidance Scenarios

As a reference, the GPC polygon clipping library [105] was chosen, a freely available (and known to be efficient) polygon clipping library. It can be also used to implement the underlying geometric computations for the situation-analysis systems. Naturally, it does not support conservative rounding and simplifications. Hence, the comparison cannot be really fair. The first column shows the run-time for 1000 runs with this library.

The second and third columns list the run-times for the developed polygon-processing library. The scenarios are run twice: once with simplifications disabled and once with simplifications enabled. It is seen that the GPC library is significantly faster for small examples. For larger examples however, the EPP library tends to be more efficient, especially with enabled simplifications. This is due to the fact that the map overlay algorithm has rather high fixed costs that are only amortised with bigger problem sizes.

Chapter 8

Summary

This thesis described the development of a polygon-processing library for safety-critical embedded systems. In this scope, three main problems have been tackled:

First, algorithms that solve fundamental problem of polygon processing like convex hull or map overlay were refined. Starting from applications like motion planning and collision detection, basic geometric objects and predicates of analytical geometry are constructed. Three-valued logic and primitives are proposed as fundamental building blocks of dependable algorithms to be used in embedded systems. The use of a third truth value allows one to make degenerate cases explicit so that they can be appropriately handled by different algorithms. Since they are succinctly described by three-valued predicates, the derived algorithms are both robust and compact.

A new solution to the map overlay, which can handle all degenerate cases, was explained in detail. Due to its flexible structure, it can be applied for various other problems like the computation of Boolean functions on polygons.

The second problem that was tackled is the verification of polygon-processing algorithms. Analytic geometry and the kernel of the software library were formalised in the interactive theorem prover HOL. Subsequently, the HOL system was used to reason about the geometric primitives in order to assure consistent definitions. Various proof tools were created, among them a comprehensive theorem library for the defined primitives as well as the integration of tactics based on dependent objects and geometric transformations. With this framework, computational geometry algorithms were verified so that required correctness properties can be formally proven.

Third, implementation problems were considered that are posed by the limited precision of arithmetic and limited resources of embedded systems. Based on a map overlay algorithm two significant extensions were made: First, conservative rounding and an algorithm were defined that take care of limited precision arithmetic to comply with the safety requirements of the system. The results are to a large extent independent of the numeric precision of the underlying microprocessors. Second, conservative map simplification limits the time and space complexity of the map overlay algorithm to guarantee the execution with predefined execution time and memory limits.

Finally, all develeped elements were grouped in several layers that represent different levels of abstraction. They do not only divide the whole system into manageable parts but make it possible to model problems and reason about them at the appropiate level of abstraction. Nevertheless, some limitations should be mentioned that remain after the work:

First, the verification could be much more detailed. Only the design of the algorithms is verified in this thesis. Although the primitives are designed to keep the distance between formalisation and code as small as possible, there can be errors in the translation to the C code, beside the usual implementation errors. For a safety-critical system, the situation would be improved if some verification is done on the implementation level, which is however not possible with the current state of art.

Second, more data structures and computational geometry algorithms can be integrated. In this thesis, all algorithms up to the network overlay were integrated in HOL system, whereas the map overlay algorithm was only partially formalised. A full formalisation of a map overlay algorithm would need more infrastructure that is currently not available in the HOL system. A lot of tedious work to formalise the data structures and the underlying theory of planar graphs would be needed for a complete integration in the theorem prover. The work here restricts to only model and verify some crucial parts of the algorithm.

Third, verification is still a very time-consuming task, which can be only accomplished by expert users of the HOL system. The constructions of proofs requires a deep knowledge of both the domain and theorem proving, which is not very widespread.

References

[1] R. D. Andreev. Algorithm for clipping arbitrary polygons. In *Computer Graphics Forum 8*, pages 183–191, 1989.

[2] F. Avnaim, J.-D. Boissonat, O. Devillers, and F.P. Preparata. Evaluating signs of determinants using single-precision arithmetic. Research Report 2306, Institut National de Recherche en Informatique et en Automatique (INRIA), Sophia-Antipolis, France, 1994.

[3] Ju. Basch, J. Erickson, L.J. Guibas, J. Hershberger, and L. Zhang. Kinetic collision detection between two simple polygons. *Computational Geometry Theory and Applications*, 27(3):211–235, 2004.

[4] J.L. Bentley and T.A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, 28(9):643–647, 1979.

[5] G. Berry. The constructive semantics of pure Esterel. http://www-sop.inria.fr/esterel.org, July 1999.

[6] J. Blinn. A trip down the graphics pipeline: Line clipping. *IEEE Computer Graphics and Applications*, 11(1):98–105, 1991.

[7] L. Bolc and P. Borowik. *Many-Valued logics*. Springer, 1992.

[8] J. Brandt and K. Schneider. Dependable polygon-processing algorithms for safety-critical embedded systems. In L.T.Yang, M. Amamiya, Z. Liu, M. Guo, and F.J. Rammig, editors, *International Conference on Embedded and Ubiquitous Computing (EUC)*, volume 3824 of *LNCS*, pages 405–417, Nagasaki, Japan, 2005. Springer.

[9] J. Brandt and K. Schneider. Using three-valued logic to specify and verify algorithms of computational geometry. In K.-K. Lau and R. Banach, editors, *International Conference on Formal Engineering Methods (ICFEM)*, volume 3785 of *LNCS*, pages 405–420, Manchester, UK, 2005. Springer.

[10] J. Brandt and K. Schneider. Efficient map overlay for safety-critical embedded systems. In *IEEE Symposium on Industrial Embedded Systems*, Antibes, France, 2006.

[11] J. Brandt and K. Schneider. *Handbook on Mobile and Ubiquitous Computing: Innovations and Perspectives*, chapter A Verified Polygon-Processing Library for Safety-Critical Embedded Systems. World Scientific Publishers, 2007. To appear.

[12] R.E. Bryant. A switch level model and simulator for MOS digital systems. *IEEE Transactions on Computers*, C-33(2):160–177, February 1984.

[13] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[14] J.A. Brzozowski and C.-J.H. Seger. *Asynchronous Circuits*. Springer, 1995.

[15] H. Brönnimann, I.Z. Emiris, V.Y. Pan, and S. Pion. Computing exact geometric predicates using modular arithmetic with single precision. In *Annual Symposium on Computational Geometry*, pages 174–182. ACM Press, 1997.

[16] C. Burnikel, K. Mehlhorn, and S. Schirra. On degeneracy in geometric computations. In *Symposium on Discrete Algorithms (SODA)*, pages 16–23, Arlington, Virginia, USA, 1994. ACM.

[17] P.A. Burrough. *Principles of Geographical Information Systems for Land Resources Assessment*. Oxford University Press, New York, 1986.

[18] K. Bühler, E. Dyllong, and W. Luther. Reliable distance and intersection computation using finite precision geometry. In R. Alt, A. Frommer, R. Baker Kearfott, and W. Luther, editors, *Numerical Software with Result Verification*, volume 2991 of *LNCS*, pages 160–190, Dagstuhl Castle, Germany, 2004. Springer.

[19] A. Camilleri, M.J.C. Gordon, and T.F. Melham. Hardware verification using higher order logic. In D. Borrione, editor, *Working Conference From HDL Descriptions to Guaranteed Correct Circuit Designs*, pages 41–66, Amsterdam, 1986. North Holland.

[20] CGAL: Computational geometry algorithms library, April 2007. `http://www.cgal.org/`.

[21] Abraham Charnes. Optimality and degeneracy in linear programming. *Econometrica*, 20(2):160–170, 4 1952.

[22] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. *Journal of the ACM*, 39(1):1–54, 1992.

[23] Bernard Chazelle and David P. Dobkin. Intersection of convex objcts in two and three dimensions. *Journal of the Assocication of Computing Machinery*, 34(1):1–27, January 1987.

[24] S.-C. Chou. A method for the mechanical derivation of formulas in elementary geometry. *Journal of Automated Reasoning*, 3:291–299, 1987.

[25] S.-C. Chou. An introduction to wu's method for mechanical theorem proving in geometry. *Journal of Automated Reasoning*, 4:237–267, 1988.

[26] S.-C. Chou. *Mechanical Geometry Theorem Proving. Mathematics and its Applications*. D. Reidel Publishing Company, Dordrecht, Boston, Lancaster, Tokyo, 1988.

[27] S-C. Chou. Automated reasoning in geometries using the characteristic set method and Gröbner basis method. In *International Symposium on Symbolic and Algebraic Computation*, pages 255–260. ACM Press, 1990.

[28] S.-C. Chou and William Schelter. Proving geometry theorems with rewrite rules. *Journal of Automated Reasoning*, 2:253–273, 1986.

[29] S.C. Chou, X.S. Gao, and J.Z. Zhang. *Machine Proofs in Geometry*. World Scientific, Singapore, 1994.

[30] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[31] K.C. Clarke. *Analytical and Computer Cartography*. Prentice Hall, Englewood Cliffs, NJ, 2nd edition edition, 1995.

[32] G. E. Collins. Quantifier elimination for real closed fields by cylindrial algebraic decomposition. In *LNCS*, volume 33, pages 134–324. Springer, 1975.

[33] P. Comninos. *Mathematical and Computer Programming Techniques for Computer Graphics*. Springer-Verlag, 2005.

[34] Coq Proof Assistant, April 2007. `http://coq.inria.fr/`.

[35] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT, 1990.

[36] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry*. Springer, 2000.

[37] M. de Berg, M. van Kreveld, and S. Schirra. A new approach to subdivision simplification. In *Twelfth International Symposium on Computer-Assisted Cartography*, volume 4, pages 79–88, Charlotte, North Carolina, 1995.

[38] O. Devillers and P. Guigue. Inner and outer rounding of set operations on lattice polygonal regions. *Computational Geometry: Theory and Applications*, 33(1–2):3–17, January 2006.

[39] H. Edelsbrunner and E.P. Mücke. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. *ACM Transactions on Graphics*, 9(1):66–104, 1990.

[40] Herbert Edelsbrunner. Edge-skeletons in arrangements with applications. *Algorithmica*, 1(1):93–109, 1986.

[41] Herbert Edelsbrunner, David G. Kirkpatrick, and Hermann A. Maurer. Polygonal intersection searching. Technical Report F64, Technische Universität Graz, February 1981.

[42] Herbert Edelsbrunner and Roman Waupotitsch. Computing a ham-sandwich cut in two dimensions. *Journal of Symbolic Computation*, 2(2):171–178, June 1986.

[43] I. Emiris and J. Canny. An efficient approach to removing geometric degeneracies. In *Symposium on Computational Geometry*, pages 74–82, Berlin, Germany, 1992. ACM Press.

[44] I. Emiris and J. Canny. A general approach to removing degeneracies. *SIAM Journal of Computing*, 24(3):650–664, June 1995.

[45] I. Emiris, J. Canny, and R. Seidel. Efficient perturbations for handling geometric degeneracies. *Algorithmica*, 19(1-2):219–242, 1997.

[46] R. Estkowski and J.S.B. Mitchell. Simplifying a polygonal subdivision while keeping it simple. In *Annual Symposium on Computational Geometry*, pages 40–49. ACM Press, 2001.

[47] C. Falls, Y. Liu, J. Snoeyink, and D. Souvaine. Testing shortcuts to maintain simplicity in subdivision simplification. In *Canadian Conference on Computational Geometry (CCCG'05)*, pages 35–38, 2005.

[48] I.D. Faux and M.J. Pratt. *Computational Geometry for Design and Manufacture*. Ellis Horwood, Chichester, U.K., 1979.

[49] J.D. Foley, A. van Dam, S.K. Feiner, and J.F. Hughes. *Computer Graphics: Principles and Practice*. Addison Wesley, 2000.

[50] A. R. Forrest. *Fundamental Algorithms for Computer Graphics*, chapter Computational geometry in practice, pages 707–724. Springer-Verlag, Berlin, Germany, 1985.

[51] S. Fortune and C. van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Transactions on Graphics*, 15(3):223–248, 1996.

[52] S. Fortune and C.J. Van Wyk. Efficient exact arithmetic for computational geometry. In *Symposium on Computational Geometry*, pages 163–172, 1993.

[53] A. Frankel, D. Nussbaum, and J.-R. Sack. Floating-point filter for the line intersection algorithm. In *Geographic Information Science*, volume 3234 of *Lecture Notes of Computer Science (LNCS)*, pages 94–105. Springer-Verlag, 2004.

[54] GNU Multiprecision Library: GNU MP, April 2007. `http://www.swox.com/gmp/`.

[55] K. Goldberg, D. Halperin, J.-C. Latombe, and R. Wilson, editors. *Algorithmic Foundations of Robotics*. A.K. Peters, Wellesley, MA, 1995.

[56] M.T. Goodrich, L.J. Guibas, J. Hershberger, and P.J. Tanenbaum. Snap rounding line segments efficiently in two and three dimensions. In *Symposium on Computational Geometry*, pages 284–293, 1997.

[57] M. Gordon, Matt Kaufmann, and Warren Hunt. Connecting HOL to ACL2. Unpublished Paper. Available at http://www.cl.cam.ac.uk/ mjcg/hol2acl2/.

[58] M.J.C. Gordon. Proving a computer correct. Technical Report TR 42, Computer Laboratory, University of Cambridge, 1983.

[59] M.J.C. Gordon. Introduction to the HOL system. In M. Archer, J.J. Joyce, K.N. Levitt, and P.J. Windley, editors, *Theorem Proving in Higher Order Logic*, pages 2–3, Davis, California, 1991. IEEE Computer Society.

[60] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[61] M.J.C. Gordon, R. Milner, and C.P. Wadsworth. *A Mechanized Logic of Computation*, volume 78 of *LNCS*. Springer, New York, 1979.

[62] G. Greiner and K. Hormann. Efficient clipping of arbitrary polygons. *ACM Transactions on Graphics (TOG)*, 17(2):71–83, 1998.

[63] M. Grimmer, K. Petras, and N. Revol. Multiple precision interval packages: Comparing different approaches. In R. Alt, A. Frommer, R. Baker Kearfott, and W. Luther, editors, *Numerical Software with Result Verification*, volume 2991 of *LNCS*, pages 64–90, Dagstuhl Castle, Germany, 2004. Springer.

[64] B. Grégoire and A. Mahboubi. Proving equalities in a commutative ring done right in Coq. In J. Hurd and T. Melham, editors, *Theorem Proving in Higher Order Logic*, volume 3603 of *LNCS*, pages 98–113, Oxford, UK, 2005. Springer.

[65] L.J. Guibas and D.H. Marimont. Rounding arrangements dynamically. *International Journal of Computational Geometry and Applications*, 8(2):157–176, 1998.

[66] P. Guige. *Constructions géometrique à précision fixée*. PhD thesis, Université de Nice-Sophia Antipolis, 2003.

[67] E. Haines and T. Möller. A collection of robust and reliable intersection tests. *Journal of Graphics Tools*, 2(4):25–44, 1997.

[68] D. Halperin and E. Packer. Iterated snap rounding. *Compututational Geometry Theory and Applications*, 23(2):209–225, 2002.

[69] R. Harper. Programming in standard ML. Carnegie Mellon University, 2004.

[70] J. Harrison. Constructing the real numbers in HOL. In L.J.M. Claesen and M.J.C. Gordon, editors, *Theorem Proving in Higher Order Logic*, pages 145–164, Leuven, Belgium, 1992. North Holland/Elsevier.

[71] J. Harrison. A HOL decision procedure for elementary real algebra. In J.J. Joyce and C.-J.H. Seger, editors, *Theorem Proving in Higher Order Logic*, volume 780 of *LNCS*, pages 426–435, Vancouver, Canada, 1994. Springer.

[72] J. Harrison. A HOL theory of euclidean space. In J. Hurd and T. Melham, editors, *Theorem Proving in Higher Order Logic*, volume 3603 of *LNCS*, pages 114–129, Oxford, UK, 2005. Springer.

[73] Paul S. Heckbert and Pat Hanrahan. Beam tracing polygonal objects. *Computer Graphics*, 18(3):119–127, July 1984.

[74] D. Hilbert. *Foundations of Geometry*. Open Court Publishing Company, La Salla, Illinois, 1971.

[75] J. Hobby. Practical segment intersection with finite precision output. *Computational Geometry*, 13(4):199–214, 1993.

[76] C. Hoffmann. *Geometric and Solid Modeling*. Morgan Kaufmann, San Mateo, CA, 1989.

[77] HOL Kananaskis, April 2007. `http://hol.sourceforge.net`.

[78] P. V. Homeier. Quotient types. Category B paper at International Conference on Theorem Proving in Higher Order Logic, 2001.

[79] J.E. Hopcroft, J.T. Schwartz, and M. Sharir. *Planning, Geometry, and Complexity of Robot Motion*. Ablex Publishing, Norwood, NJ, 1987.

[80] K. Hormann and A. Agathos. The point in polygon problem for arbitrary polygons. *Computational Geometry: Theory and Applications*, 20(3):131–144, 2001.

[81] Isabelle, April 2007. `http://www.cl.cam.ac.uk/research/hvg/Isabelle/`.

[82] T. Jebelean. Rational arithmetic using FPGAs. In Will Moore and Wayne Luk, editors, *More FPGAs*, pages 262–273, Abingdon, England, 1993. Abingdon EE&CS Books.

[83] S. Kalvala. HOL around the world. In M. Archer, J.J. Joyce, K.N. Levitt, and P.J. Windley, editors, *Theorem Proving in Higher Order Logic*, pages 4–12, Davis, California, 1991. IEEE Computer Society.

[84] S.C. Kleene. *Introduction to Metamathematics*. North Holland, 1952.

[85] D.E. Knuth. *Axioms and Hulls*, volume 606 of *LNCS*. Springer, 1992.

[86] D.E. Knuth. *The Art of Computer Programming*, volume 2. Addison Wesley, 1998.

[87] J.-C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, Boston, 1991.

[88] J.P. Laumond and M.H. Overmars, editors. *Algorithms for Robotic Motion and Manipulation*. A.K. Peters, Wellesley, MA, 1996.

[89] You-Dong Liang and Brian A. Barsky. An analysis and algorithm for polygon clipping. *Communications of the ACM*, 26(11):868–877, 1983.

[90] M. Lin and S. Gottschalk. Collision detection between geometric models: A survey. In *IMA Conference on Mathematics of Surfaces*, pages 37–56, 1998.

[91] M.C. Lin. *Efficient Collision Detection for Animation and Robotics*. PhD thesis, Department of Electrical Engineering, University of California at Berkeley, 1993.

[92] P. Lindstrom and G. Turk. Fast and memory efficient polygonal simplification. In *Conference on Visualization '98*, pages 279–286. IEEE Computer Society Press, 1998.

[93] T. Lozano-Perez and M.A. Wesley. An algorithm for planning collision-free paths among polyhedral obstacles. *Commun. ACM*, 22(10):560–570, 1979.

[94] S. Malik. Analysis of cyclic combinational circuits. In *Conference on Computer Aided Design (ICCAD)*, pages 618–625, Santa Clara, California, November 1993. IEEE Computer Society.

[95] P. Marwedel. *Embedded System Design*. Springer-Verlag New York, Secaucus, NJ, 2006.

[96] P.J. McKerrow. *Introduction to Robotics*. Addison-Wesley, Reading, MA, 1991.

[97] K. Mehlhorn. *Multi-Dimensional Searching and Computational Geometry*, volume 3 of *Data Structures and Algorithms*, chapter VIII. Springer, 1984.

[98] K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[99] V. Milenkovic. Shortest path geometric rounding. *Algorithmica*, 27(1):57–86, 2000.

[100] V. J. Milenkovic. *Verifiable Implementations of Geometric Algorithms Using Finite Precision Arithmetic*. PhD thesis, Carnegie Mellon University, July 1988.

[101] C. Montani and Michele Re. Vector and raster hidden-surface removal using parallel connected stripes. *IEEE Computer Graphics and Application*, 7(7):14–23, July 1987.

[102] M. Moore and J. Wilhelms. Collision detection and response for computer animation. In *Annual Conference on Computer graphics and Interactive Techniques*, pages 289–298, New York, NY, USA, 1988. ACM Press.

[103] M.E. Mortenson. *Geometric Modeling*. Wiley, New York, 1985.

[104] Moscow ML, April 2007. `http://www.dina.kvl.dk/~sestoft/mosml.html`.

[105] A. Murta. GPC. `http://www.cs.man.ac.uk/aig/staff/alan/software/gpc.html`, April 2007.

[106] H. Müller and S. Abramowski. A numerically stable polygon clipping algorithm. Interner Bericht 12, Universität Karlsruhe, May 1984.

[107] J. Narboux. A decision procedure for geometry in Coq. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics*, volume 3223 of *LNCS*, pages 225–240, Park City, Utah, USA, 2004. Springer.

[108] T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[109] NuPRL, April 2007. `http://www.cs.cornell.edu/Info/Projects/NuPRL/`.

[110] L.C. Paulson. *ML for the Working Programmer*. Cambridge University, 1991.

[111] D. Pichardie and Y. Bertot. Formalizing convex hull algorithms. In R.J. Boulton and P.B. Jackson, editors, *Theorem Proving in Higher Order Logics*, volume 2152 of *LNCS*, pages 346–361, Edinburgh, Scotland, UK, 2001. Springer.

[112] F.P. Preparata and M.I. Shamos. *Computational geometry: an introduction*. Springer-Verlag New York, Inc., 1985.

[113] D.M. Priest. Algorithms for arbitrary precision floating point arithmetic. In P. Kornerup and D.W. Matula, editors, *Symposium on Computer Arithmetic*, pages 132–144. IEEE Computer Society, 1991.

[114] F. Puitg and J.-F. Dufourd. Formal specification and theorem proving breakthroughs in geometric modeling. In J. Grundy and M.C. Newey, editors, *Theorem Proving in Higher Order Logic*, volume 1479 of *LNCS*, pages 401–422, Canberra, Australia, 1998. Springer.

[115] A. Rappoport. An efficient algorithm for line and polygon clipping. *The Visual Computer: International Journal of Computer Graphics*, 7(1):19–28, 1991.

[116] T.W. Reps, M. Sagiv, and R. Wilhelm. Static program analysis via 3-valued logic. In R. Alur and D.A. Peled, editors, *Conference on Computer Aided Verification (CAV)*, volume 3114 of *LNCS*, pages 15–30, Boston, MA, USA, 2004. Springer.

[117] R.J. Schilling. *Fundamentals of Robotics, Analysis and Control*. Prentice Hall, Englewood Cliffs, NJ, 1990.

[118] K. Schneider, J. Brandt, T. Schuele, and T. Tuerk. Improving constructiveness in code generators. In *Synchronous Languages, Applications, and Programming (SLAP)*, Edinburgh, Scotland, UK, 2005.

[119] K. Schneider, J. Brandt, T. Schuele, and T. Tuerk. Maximal causality analysis. In *Conference on Application of Concurrency to System Design (ACSD)*, pages 106–115, St. Malo, France, June 2005. IEEE Computer Society.

[120] E. Schömer and C. Thiel. Efficient collision detection for moving polyhedra. In *Annual Symposium on Computational Geometry*, pages 51–60. ACM Press, 1995.

[121] S. Sechrest and D.P. Greenberg. A visible polygon reconstruction algorithm. *Computer Graphics*, 15(3):17–27, August 1981.

[122] R. Sedgewick. *Algorithms*. Addison-Wesley, 2nd edition, 1988.

[123] J.R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete and Computational Geometry*, 18(3):305–363, October 1996.

[124] T.R. Shiple. *Formal Analysis of Synchronous Circuits*. PhD thesis, University of California at Berkeley, 1996.

[125] F. Somenzi. Binary decision diagrams. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*, volume 173 of *NATO Science Series F: Computer and Systems Sciences*, pages 303–366. IOS Press, 1999.

[126] Z. Sun, G. Bebis, and R. Miller. On-road vehicle detection: A review. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(5):694–711, 2006.

[127] I.E. Sutherland and G.W. Hodgeman. Reentrant polygon clipping. *Communications of the ACM*, 17(1):32–42, 1974.

[128] A. Tarski. *A Decision Method for Elementary Algebra and Geometry*. University of California, 1951.

[129] F. Vahid and T. Givargis. *Embedded System Design: A Unified Hardware/Software Introduction*. John Wiley and Sons, 2002.

[130] B.R. Vatti. A generic solution to polygon clipping. *Communications of the ACM*, 35(7):56–63, 1992.

[131] K. Weiler and P. Atherton. Hidden surface removal using polygon area sorting. In *Annual Conference on Computer Graphics and Interactive Techniques*, pages 214–222. ACM Press, 1977.

[132] W.-T. Wu. On the decision problem and the mechanization of theorem proving in elementary geometry. *Scientia Sinica*, 21:157–179, 1978.

[133] W.-T. Wu. Basic principles of mechanical theorem proving in geometries. *Journal of Systems Science and Mathematical Science*, 4(3):207–235, 1984.

[134] W.-T. Wu. Basic principles of mechanical theorem proving in geometries. *Journal of Automated Reasoning*, 2(4):221–252, 1986.

[135] C. Yap. Robust geometric computation. In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry*. CRC Press, 1997.

[136] C. Yap. Towards exact geometric computation. *Computational Geometry Theory Applications*, (7):3–23, 1997.

[137] C. K. Yap. A geometric consistency theorem for a symbolic perturbation scheme. In *Annual Symposium on Computational Geometry*, pages 134–142, New York, NY, USA, 1988. ACM Press.

[138] C.-K. Yap. Symbolic treatment of geometric degeneracies. *Journal of Symbolic Computation*, 10(3–4):349–370, 1990.

[139] J.-Z. Zhang, S.-C. Chou, and X.-S. Gao. Automated production of traditional proofs for theorems in euclidean geometry. *Annals of Mathematics and Artificial Intelligence*, 13(1–2), 3 1995.

Chapter A

Curriculum Vitae

Persönliche Daten

Name	Jens Brandt
Geburtsdatum	12. Januar 1978
Geburtstort	Mainz
Familienstand	ledig
Staatsangehörigkeit	deutsch

Schulbildung

08/1984 – 07/1988	Grundschule Stromberg / Hunsrück
08/1988 – 06/1997	Stefan-George-Gymnasium (Bingen / Rhein) Abschluss: Abitur

Zivildienst

09/1997 – 09/1998	Pflegedienst im Altenheim Martin-Luther-Stift in Bingen / Rhein

Hochschulstudium

10/1998 – 09/2003	Studium an der TU Kaiserslautern Angewandte Informatik Abschluss: Diplom-Informatiker
seit 09/2003	Studium an der TU Kaiserslautern Promotionsprogramm Informatik