Using Three-Valued Logic to Specify and Verify Algorithms of Computational Geometry

Jens Brandt and Klaus Schneider

University of Kaiserslautern,
Reactive Systems Group, Department of Computer Science,
P.O. Box 3049, 67653 Kaiserslautern, Germany
http://rsg.informatik.uni-kl.de

Abstract. Many safety-critical systems deal with geometric objects. Reasoning about the correctness of such systems is mandatory and requires the use of basic definitions of geometry for the specification of these systems. Despite the intuitive meaning of such definitions, their formalisation is not at all straightforward: In particular, degeneracies lead to situations where none of the Boolean truth values adequately defines a geometric primitive. Therefore, we use a three-valued logic for the definition of geometric primitives to explicitly handle such degenerate cases. We have implemented a three-valued library of linear geometry in an interactive theorem prover for higher order logic which allows us to specify and verify entire algorithms of computational geometry.

1 Introduction

Many applications like motion planning in robotics or collision detection of autonomous vehicles have to consider the positions of physical objects in their environment. For most of these applications, it is sufficient to model the considered objects as polygons (or polyhedra) in an Euclidian plane (or space). This way, these applications directly rely on algorithms of computational geometry [6]. In particular, basic geometric primitives are used to develop software systems for controlling the spatial behaviour of physical objects like autonomous vehicles.

Although these algorithms are not new, their use in upcoming safety-critical embedded systems, used e.g. in automobiles, calls for a more rigorous treatment to guarantee the correctness for all possible inputs. To this end, formal definitions of geometric objects and primitives are required to specify and verify fundamental algorithms of computational geometry. At a first glance, the definition of geometric primitives appears to be easy, since they can be depicted in a natural and intuitive way. However, even definitions of simple geometric primitives are not at all straightforward, since they have to cover all possible cases: For example, what is the intersection point of two line segments, if both line segments are identical or share a common endpoint? Such degenerate cases [6] make clear that consistent definitions, that have to hold for all algorithms, are subtle. In fact, many algorithms only work under certain preconditions on the inputs as e.g. that all input points are pairwise distinct, or that no three input points are collinear.

Even worse, we found that for some primitives, there is no ‘good’ definition at all. For example, to define whether a point on the edge of a polygon belongs to the interior

K.-K. Lau and R. Banach (Eds.): ICFEM 2005, LNCS 3785, pp. 405–420, 2005.

or not, leads to problems in one or the other algorithm. In our opinion, the best solution is to make these degenerate cases explicit so that definitions and algorithms can directly handle them in a way that is appropriate for the context algorithm. To this end, we employ a three-valued logic to define geometric primitives. In the abovely mentioned example, we can express that a point is inside, outside, or on the edge of a polygon.

Moreover, we propose the use of higher order logics and corresponding theorem provers to consistently reason about the correctness of geometric algorithms. To this end, we extended the HOL theorem prover [10] by a library on two-dimensional analytic geometry that consists of three-valued geometric primitives. This library is not only useful to reason more efficiently about algorithms using two-valued primitives, it may also be viewed as the core of a software library for three-valued computational geometry [2] that is sometimes more concise than the corresponding two-valued version.

Our work deviates from previous work on reasoning about geometric problems with theorem provers in several ways: In particular, Wu's work [18] on translating geometric propositions to an algebraic form, i.e. equations between polynomials, is well-known. Various researchers improved and finally implemented this approach. Several hundred theorems about basic geometric objects like lines, triangles, and circles have been automatically proven with these theorem provers [5]. However, this approach is limited to reason about particular instances of geometric problems, but can not be used to reason about algorithms to solve classes of geometric problems, which is our concern.

The work closest to ours is that of Pichardie and Bertot [15]: Based on the work of Knuth [12], they formalised basic principles of convex hull algorithms. As in our approach, the orientation primitive (see Section 3.3) plays a central role to gain a new level of abstraction. In contrast to our work, they used two-valued logic to formalise geometric primitives. As a consequence of this, they circumvent problems of degenerate cases by modifying the orientation primitive or perturbing the input data. Furthermore, their scope is restricted to convex hull algorithms.

In this paper, we focus on the formalisation of two-dimensional, linear objects like lines, segments and polygons using three-valued geometric primitives. Due to lack of space, we only focus on the definition of the three-valued geometric primitives and show how degenerate cases are handled with appropriate definitions. Detailed definitions, in particular, the code for the HOL library, as well as further case studies like the verification of the Cohen-Sutherland clipping algorithm [9], are available on our website.

This paper is organised as follows: Section 2 describes the formalisation of analytic geometry and discusses the problem of degenerate cases in computational geometry. Section 3 presents our three-valued logic and its use for specifying geometric properties and primitives. Section 4 shows corresponding proof techniques and illustrates them with the help of a small example. Finally, Section 5 draws some conclusions.

2 Prerequisites

In mathematics, geometry is usually formalised in the vector space $\mathbb{R}^n$. However, real computers use floating point arithmetic of a limited precision, so that rounding errors appear as further problems. To circumvent these problems, we use rational numbers of arbitrary precision. The use of rational numbers is motivated by the observation that

most algorithms only deal with linear objects like lines and polygons, so that all operations can be performed on rational numbers. As the HOL system does not directly provide rational numbers, we formalised them on our own.

2.1 Formalisation of Basic Analytic Geometry

Since we investigate problems of two-dimensional geometry, a vector is given by an ordered pair of rationals (`rat#rat`), encapsulated in a new type `vec`.

Definitions. For this type, we make the following definitions: $\mathbf{0}$ denotes the zero vector, and $\mathbf{ux}$ and $\mathbf{uy}$ denote the unit vectors. The components of a vector v can be accessed by x_v and y_v, respectively. A vector can be mirrored, rotated and multiplied by a scalar. A pair of vectors can be added and subtracted.

$$\begin{aligned}
\texttt{vec_mir_def} &\vdash_{\text{def}} \mathsf{mir}(v_1) = (\,-\mathsf{x}_{v_1}\,;\,-\mathsf{y}_{v_1}\,)\\
\texttt{vec_orth_def} &\vdash_{\text{def}} \mathsf{orth}(v_1) = (\,\mathsf{y}_{v_1}\,;\,-\mathsf{x}_{v_1}\,)\\
\texttt{vec_scale_def} &\vdash_{\text{def}} r_1 \cdot v_1 = (\,r_1 \cdot \mathsf{x}_{v_1}\,;\,r_1 \cdot \mathsf{y}_{v_1}\,)\\
\texttt{vec_add_def} &\vdash_{\text{def}} v_1 + v_2 = (\,\mathsf{x}_{v_1} + \mathsf{x}_{v_2}\,;\,\mathsf{y}_{v_1} + \mathsf{y}_{v_2}\,)\\
\texttt{vec_sub_def} &\vdash_{\text{def}} v_1 - v_2 = (\,\mathsf{x}_{v_1} - \mathsf{x}_{v_2}\,;\,\mathsf{y}_{v_1} - \mathsf{y}_{v_2}\,)
\end{aligned}$$

Multiplication of vectors is not uniquely defined: In addition to the dot product, the cross product is well-known. For the two-dimensional case, it does not exist per se, but a related product that is sometimes called *perp dot product* does exist: This is the dot product where the first vector is replaced by the perpendicular vector. With its help, the linear dependency of two vectors is easily defined.

$$\begin{aligned}
\texttt{sprod_def} &\vdash_{\text{def}} v_1 \circ v_2 = x_{v_1} \cdot x_{v_2} + y_{v_1} \cdot y_{v_2}\\
\texttt{cprod_def} &\vdash_{\text{def}} v_1 \times v_2 = x_{v_1} \cdot y_{v_2} - y_{v_1} \cdot x_{v_2}\\
\texttt{lindep_def} &\vdash_{\text{def}} \mathsf{lindep}(v_1, v_2) = (v_1 \times v_2 = 0)
\end{aligned}$$

Theorems. The vectors `vec` form a vector space over the rational numbers `rat`. As consequence of this, various arithmetic laws can be derived. For example, the cross product has the following properties (inter alia):

$$\begin{aligned}
\texttt{CPROD_RDISTRIB} &\vdash (v_1 + v_2) \times v_3 = (v_1 \times v_3) + (v_2 \times v_3)\\
\texttt{CPROD_RSUM} &\vdash v_1 \times (v_1 + v_2) = v_1 \times v_2
\end{aligned}$$

Clearly, our library 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.

$$\begin{aligned}
\texttt{VEC_CRAMERS_RULE} \vdash\ & \neg(v_1 \times v_2 = 0) \rightarrow ((v_0 = r_1 \cdot v_1 + r_2 \cdot v_2) =\\
& (r_1 = (v_0 \times v_2)/(v_1 \times v_2)) \wedge (r_2 = (v_1 \times v_0)/(v_1 \times v_2))\,)
\end{aligned}$$

We have also proved that the linear dependency relation is an equivalence relation and that it commutes with various vector operations.

2.2 Degenerate Cases

Most algorithms of computational geometry are designed for the 'general case': Depending on the algorithm, several preconditions are assumed, e.g. no points coincide, given lines are not parallel, or that no three lines intersect in a common point. Thus, so-called *degenerate cases* pose a well-known problem to algorithms in computational geometry [7, 14].

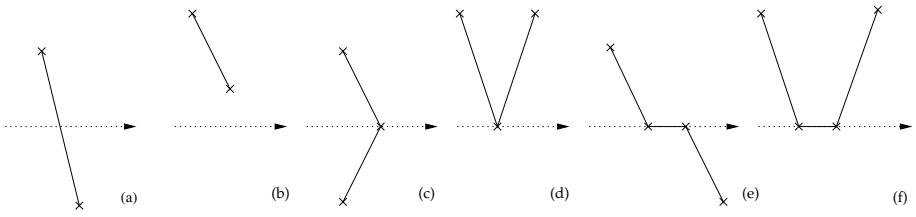

Fig. 1. Cases of the parity algorithm

As an example, consider the *parity algorithm*, which determines whether a point p is inside or outside of a polygon P: It counts the intersections of an arbitrary ray starting in p with edges of the polygon P. If the number of intersections is odd, p is inside; otherwise p is outside P. Figure 1 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 any problems. (c) to (f) show degenerate cases where either a vertex or an edge of the polygon is on the ray. To make the parity algorithm work correctly, we have to define some of the cases as intersections: cases (c) and (e) are intersections, whereas cases (d) and (f) are not.

Degenerate cases like the above mentioned ones require a substantial amount of additional effort. Since there are numerous degenerate cases, it is not recommended to directly address them in the algorithms as special cases. Instead, some other methods have been proposed that we briefly discuss in the remainder of this section.

Symbolic Perturbations. A popular method to handle degenerate cases is the symbolic perturbation of degenerate inputs [7], which resolves degeneracies by simply hiding them (black box method). Intuitively, each geometric coordinate is replaced with a symbolically perturbed coordinate, given by a polynomial of an infinitesimal small number ϵ. Substitution of the symbolically perturbed coordinates in a primitive expression results in a polynomial in the variable ϵ with coefficients determined by the original geometric coordinates. 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. This resolves all degeneracies of the considered primitive. Programs that use this technique tend to be smaller and more robust: the tedious treatment of many special cases is replaced by a single consistent perturbation scheme.

While this method is certainly a useful tool for the implementation of geometric algorithms, existing perturbation schemes have shown not to be as applicable as desired [4]. First, symbolic perturbations give the programmer a rather unsatisfactory choice: either to find an approximation of the original problem, or 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. Second, symbolic perturbations need to be worked out in detail, a task that may be very complex. This has been done only for a few geometric primitives. Finally, objects that are constructed by the algorithm (e.g. intersection points) are often forbidden in the computation, because their perturbation

function depends on the construction of the object and is much more complicated than the one for a primitive object.

Explicit Treatment of Degeneracies. The explicit treatment of degeneracies suffers from the enormous number of cases. As an example, consider the intersection of two line segments: In general position, two segments either do not intersect or intersect at a point interior to both segments. Two intersecting segments in special position may overlap, share a common endpoint or have one segment endpoint interior to the other segment – and each case exists in various slightly different variations. Hence, it is obvious that a systematic analysis is indispensable.

As another 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' point sets). However, if we calculate the difference of two polygons 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 points. This naturally motivates the use of a three-valued logic that we explain in the following section. Three-valued logic allows us to describe geometric properties and algorithms more precisely and more compactly without enumerating many tedious cases. Note that although these cases do not disappear, three-valued logic makes it possible to handle them in a systematic and concise way.

3 Using Three-Valued Logics in Analytic Geometry

Classical mathematical logic is bivalent, i.e. there are two possible truth values: *true* and *false*. The law of the excluded middle is one of the foundations of the classical two-valued logic: A proposition P is either true or false, and there is no other choice.

In the early 1920s, the Polish philosopher and logician Jan Lukasiewicz dealt with philosophical problems like Aristotle's paradox of the sea battle. He pointed out that these problems can be solved by introducing a third value. In the following, a lot of mathematicians engaged in this domain of logics, among them Stephen C. Kleene. In the late 1930s, he introduced his three-valued logics for the analysis of partial recursive primitives [11, 1]. Within his work, the third truth value modeled situations where expressions are *undefined*.

Today, many-valued logics have found many applications in computer science. For example, they are applied to solve problems of database systems, artificial intelligence, simulation of hardware circuits, [8, 13], and program analysis [16, 17].

3.1 Three-Valued Logic Operators

Reconsider the example of the *point in polygon* at the end of Section 2.2. 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 outside, while the points on the edge are assigned U (which is interpreted as 'borderline' or generally, *degenerate case*).

	$\ddot{\neg}$
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

Fig. 2. Truth tables of basic logical operators

$\ddot{\rightarrow}$	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

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

Fig. 3. Truth tables of $\ddot{\rightarrow}$, $\ddot{\leftrightarrow}$, $txor$ and $\ddot{*}$

These considerations give rise to the definitions of the basic three-valued connectives shown in Figure 2: The negation $\ddot{\neg}$ interchanges the inside and the outside of a polygon, and all points of the edge remain on the edge. The conjunction $\ddot{\wedge}$ represents the intersection of two polygons P_1 and P_2: Points that are both in polygon P_1 and in polygon P_2 belong to the intersection. Points that are either outside P_1 or P_2 are not part of the intersection. Finally, points that are located on the edge of one polygon and not outside the other, are on the edge of the intersection. The disjunction $\ddot{\vee}$ can be derived analogously. Hence, the operators $\ddot{\neg}$, $\ddot{\wedge}$ and $\ddot{\vee}$ correspond to the basic connectives of Kleene's three-valued logic [11].

Definitions. Starting from these definitions, we introduce further operators: implication $\ddot{\rightarrow}$, equivalence $\ddot{\leftrightarrow}$, exclusive-or $\ddot{\oplus}$ and a modified conjuction $\ddot{*}$. Figure 3 gives their truth tables. While $\ddot{\rightarrow}$, $\ddot{\leftrightarrow}$ and $\ddot{\oplus}$ are defined with the help of the basic operators, $\ddot{*}$ (whose meaning will be explained in Section 3.3) is defined by its truth table.

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

We extend the theory by existential and universal quantification. To this end, we recall the disjunctive interpretation of $\exists$ and the conjunctive interpretation of $\forall$ (also known as *substitution interpretation* for finite universes) that defines $\exists x.P(x) = \bigvee_{x \in \mathcal{D}_x} P(x)$ and $\forall x.P(x) = \bigwedge_{x \in \mathcal{D}_x} P(x)$, respectively. For our HOL theory, we chose the following, more feasible definition, which corresponds to the previous one:

$$\texttt{exists3\_def} \ \vdash_{\text{def}} \ \ddot{\exists} P = \mathbf{if}\,(\exists x.P(x) = \mathsf{T})\,\mathbf{then}\,\mathsf{T}\,\mathbf{else}\,(\mathbf{if}\,(\forall x.P(x) = \mathsf{F})\,\mathbf{then}\,\mathsf{F}\,\mathbf{else}\,\mathsf{U})$$
$$\texttt{forall3\_def} \ \vdash_{\text{def}} \ \ddot{\forall} P = \mathbf{if}\,(\forall x.P(x) = \mathsf{T})\,\mathbf{then}\,\mathsf{T}\,\mathbf{else}\,(\mathbf{if}\,(\exists x.P(x) = \mathsf{F})\,\mathbf{then}\,\mathsf{F}\,\mathbf{else}\,\mathsf{U})$$

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

$\ddot{\sqsubset}$	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. 4. 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

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

Fig. 5. Truth tables of $\leq$, $\geq$ and $\twoheadrightarrow$

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. Hence, we define a relation $\ddot{\sqsubset} : \mathbb{T} \times \mathbb{T} \to \mathbb{T}$ that compares two truth values (see Figure 4). 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 4 gives the truth tables of $\ddot{\lhd}$ and $\ddot{\rhd}$.

Integrating Two-Valued and Three-Valued Propositions. Introducing three-valued formulas into a two-valued environment like HOL poses the problem of integrating both logics. First, how are two-valued terms embedded into three-valued formulas? This direction is rather simple; the definition of the required embedding operator $\ddot{\Delta} : \mathbb{B} \to \mathbb{T}$ is straightforward: *true* is mapped onto T, and *false* is mapped onto F. Second, how are three-valued formulas transformed to the Boolean domain? This 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 $\neg(P = \mathsf{F})$, we introduce two new relations $\leq$ and $\geq$ 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 5).

3.2 Geometric Objects

The definitions on vectors as given in Section 2.1 are the basis of the formalisation of geometric objects. Vectors are the basic objects of analytic geometry, which are used to define all other objects. With the exception of points (that are represented by their position vectors and thus, are equivalent to vectors), all geometric objects are formed by sets of points that are the solution of a proposition. For example, a line given by two (different) points p and q, consists of all points $(x; y)$ that are a solution of the following equation:

$$\text{line} : \exists\lambda.\ (x; y) = (x_p; y_p) + \lambda \cdot (x_q - x_p; y_q - y_p)$$

Analogously, a square with the vertices $(0;0)$, $(1;0)$, $(1;1)$ and $(0;1)$ is defined by the following inequations:

$$\text{square} : 0 < x \wedge x < 1 \wedge 0 < y \wedge y < 1$$

Using classical logic, all characteristic propositions are two-valued (as above). Thus, a point is either a solution or not, i.e. it is either part of the object or not. In contrast, we use three-valued propositions to explicitly express degenerate points. These degenerate points are related with the edges and endpoints of objects: Inequations describe two-dimensional objects and degenerates points are located on the edge of the object, i.e. at the transition between the interior and the exterior of an object. Equations generally describe one-dimensional objects with special cases located at the 'end' of these objects. In both cases, the degeneracies are an effect of inequations, which can be seen as the actual source of degeneracies.

Hence, we introduce three-valued inequations between rational numbers. In the case where the left hand side is equal to the right hand side, the validity of the inequation is *undefined*:

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

The relation $\prec$ has the following properties:

$$\begin{aligned}
\texttt{RAT_LES3_REF} &\ \vdash\ (r_1 \prec r_1) = \mathsf{U}\\
\texttt{RAT_LES3_ANTISYM} &\ \vdash\ \ddot{\neg}(r_2 \prec r_1) = (r_1 \prec r_2)\\
\texttt{RAT_LES3_TRANS} &\ \vdash\ (r_1 \prec r_2)\ \ddot{*}\ (r_2 \prec r_3)\ \twoheadrightarrow\ (r_1 \prec r_3)
\end{aligned}$$

Using this relation, we define in the following other geometric objects. We thereby focus on two-dimensional linear objects, i.e. lines, rays, segments and rectangles. Circles, curves, and objects of higher dimensions are not considered, since they are not relevant for most applications. Nevertheless, the principles that are presented in the following can be applied to them, too.

Lines, Rays and Segments. In analytic geometry, a line is usually defined by its parametric equation (see first equation in the previous section). To convert the classic definition of a line to a three-valued one, all two-valued operators are exchanged by their three-valued counterparts:

$$\text{line} : \ddot{\exists}\lambda.\ (x;y) = (x_p;y_p) + \lambda \cdot (x_q - x_p; y_q - y_p)$$

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. A ray and a line segment can be specified similarly: For the ray, we add the condition that λ must be positive, and for a line segment, λ must be greater than 0 and less than 1. With these restrictions, the starting points of these objects are degenerate points.

$$\begin{aligned}
\text{ray} &: \ddot{\exists}\lambda.\ (x;y) = (x_p;y_p) + \lambda \cdot (x_q - x_p; y_q - y_p)\ \ddot{\wedge}\ (0 \prec \lambda)\\
\text{segment} &: \ddot{\exists}\lambda.(x;y) = (x_p;y_p) + \lambda \cdot (x_q - x_p; y_q - y_p)\ \ddot{\wedge}\ (0 \prec \lambda)\ \ddot{\wedge}\ (\lambda \prec 1)
\end{aligned}$$

HOL *Theory of Lines.* In our HOL theory, lines, rays, and line segments are represented by the same type `line`. A line is represented by a pair of different vectors,

which represent the points in the parametric equation. We use the constructor $\overrightarrow{(v_1, v_2)}$ that converts two vectors v_1 and v_2 to a line (a new data type). After the construction of a line $\ell = \overrightarrow{(v_1, v_2)}$, the points v_1 and v_2 used for the construction of the line, can still be accessed by the following functions: $\mathsf{beg}(\overrightarrow{(v_1, v_2)}) := v_1$ and $\mathsf{end}(\overrightarrow{(v_1, v_2)}) := v_2$.

The following functions define the point sets of a line, a ray or a segment. They correspond to the definitions of the previous paragraph.

$\texttt{on\_line\_def} \vdash_{\text{def}}$
$\mathsf{onLine}(\ell, v) = \ddot{\exists}\lambda.\, v = \mathsf{beg}(\ell) + \lambda \cdot (\mathsf{end}(\ell) - \mathsf{beg}(\ell))$
$\texttt{on\_ray\_def} \vdash_{\text{def}}$
$\mathsf{onRay}(\ell, v) = \ddot{\exists}\lambda.\, v = \mathsf{beg}(\ell) + \lambda \cdot (\mathsf{end}(\ell) - \mathsf{beg}(\ell)) \,\ddot{\wedge}\, (0 \prec \lambda)$
$\texttt{on\_seg\_def} \vdash_{\text{def}}$
$\mathsf{onSeg}(\ell, v) = \ddot{\exists}\lambda.\, v = \mathsf{beg}(\ell) + \lambda \cdot (\mathsf{end}(\ell) - \mathsf{beg}(\ell)) \,\ddot{\wedge}\, (0 \prec \lambda) \,\ddot{\wedge}\, (\lambda \prec 1)$

3.3 Geometric Primitives

Most geometric algorithms rely on a small number of geometric primitives. Among them, there are primitives that take some input and classify it as one of a constant number of possible cases, as e.g.:

- *Position of two points.* A point p is left from a point q iff $\chi_{\mathsf{left}}(p, q) := x_q - x_p > 0$. Analogously, point p is below q iff $\chi_{\mathsf{below}}(p, q) := y_q - y_p > 0$.
- *Orientation of three points.* The points p, q and r define a left turn iff

$$\chi_{\mathsf{lturn}}(p, q, r) := \begin{vmatrix} x_p & y_p & 1 \\ x_q & y_q & 1 \\ x_r & y_r & 1 \end{vmatrix} > 0 \tag{1}$$

Degeneracies with respect to such a primitive P are inputs $\boldsymbol{x}$ that cause the characteristic function to become zero $\chi_P(\boldsymbol{x}) = 0$. Following the approach presented in Section 3, the result U is returned in these cases.

Three-Valued Primitives. To define the primitives, we use the three-valued less-than relation $\prec$ of the previous section. Since all primitives of the previous section compare their result with zero, we additionally introduce the following predicate:

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

With their help, primitives for determining whether one point is on the left or below of another point are defined as follows:

$\texttt{left\_def} \;\vdash_{\text{def}}\; \mathsf{left}(v_1, v_2) = \mathsf{pos}(\mathsf{x}_{v_2} - \mathsf{x}_{v_1})$
$\texttt{below\_def} \;\vdash_{\text{def}}\; \mathsf{below}(v_1, v_2) = \mathsf{pos}(\mathsf{y}_{v_2} - \mathsf{y}_{v_1})$

Note that these primitives are three-valued. The following theorems prove some sort of reflexivity, antisymmetry and transitivity laws.

$\texttt{LEFT\_REF} \;\vdash\; \mathsf{left}(v_1, v_1) = \mathsf{U}$
$\texttt{LEFT\_ASYM} \;\vdash\; \mathsf{left}(v_1, v_2) = \ddot{\neg}\mathsf{left}(v_2, v_1)$
$\texttt{LEFT\_TRANS} \;\vdash\; \mathsf{left}(v_1, v_2) \mathbin{\ddot{*}} \mathsf{left}(v_2, v_3) \;\twoheadrightarrow\; \mathsf{left}(v_1, v_3)$

LEFT_TRANS makes use of the connectives $\ddot{*}$ and $\twoheadrightarrow$, which usually appear together in a proposition. They allow a succinct description of the following cases:

- If both $\mathsf{left}(v_1, v_2) = \mathsf{T}$ and $\mathsf{left}(v_2, v_3) = \mathsf{T}$, then $\mathsf{left}(v_1, v_3) = \mathsf{T}$.
- If $\mathsf{left}(v_1, v_2) = \mathsf{T}$ and $\mathsf{left}(v_2, v_3) = \mathsf{U}$ or vice versa, then $\mathsf{left}(v_1, v_3) = \mathsf{T}$.
- If $\mathsf{left}(v_1, v_2) = \mathsf{U}$ and $\mathsf{left}(v_2, v_3) = \mathsf{U}$, then $\mathsf{left}(v_1, v_3) = \mathsf{U}$.
- If $\mathsf{left}(v_1, v_2) = \mathsf{F}$ or $\mathsf{left}(v_2, v_3) = \mathsf{F}$, then nothing is said about $\mathsf{left}(v_1, v_3)$.

The orientation primitives can be defined analogously:

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

Again, various properties are proven for the orientation primitive:

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

$$\texttt{LTURN\_TRIAN} \vdash \mathsf{lturn}(v_1, v_2, v_4) \mathbin{\ddot{*}} \mathsf{lturn}(v_2, v_3, v_4) \mathbin{\ddot{*}} \mathsf{lturn}(v_3, v_1, v_4) \twoheadrightarrow \mathsf{lturn}(v_1, v_2, v_3)$$
$$\texttt{LTURN\_TRANS} \vdash (\mathsf{lturn}(v_1, v_2, v_3) \mathbin{\ddot{\wedge}} \mathsf{lturn}(v_1, v_2, v_4) \mathbin{\ddot{\wedge}} \mathsf{lturn}(v_1, v_2, v_5) \geq \mathsf{U}) \rightarrow \mathsf{lturn}(v_1, v_3, v_4) \mathbin{\ddot{*}} \mathsf{lturn}(v_1, v_4, v_5) \twoheadrightarrow \mathsf{lturn}(v_1, v_3, v_5)$$
$$\texttt{LTURN\_MOD1} \vdash (\mathsf{onRay}(\overrightarrow{(v_2, v_3)}, v_4) = \mathsf{T}) \rightarrow \mathsf{lturn}(v_1, v_2, v_3) = \mathsf{lturn}(v_1, v_2, v_4)$$
$$\texttt{LTURN\_MOD2} \vdash (\mathsf{onRay}(\overrightarrow{(v_4, v_3)}, v_2) = \mathsf{T}) \rightarrow \mathsf{lturn}(v_1, v_2, v_4) = \mathsf{lturn}(v_1, v_3, v_4)$$

These theorems are three-valued reformulations of the ones that can be found in [15]. 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 6 (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 left-turn primitive 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 6 (b)). The last

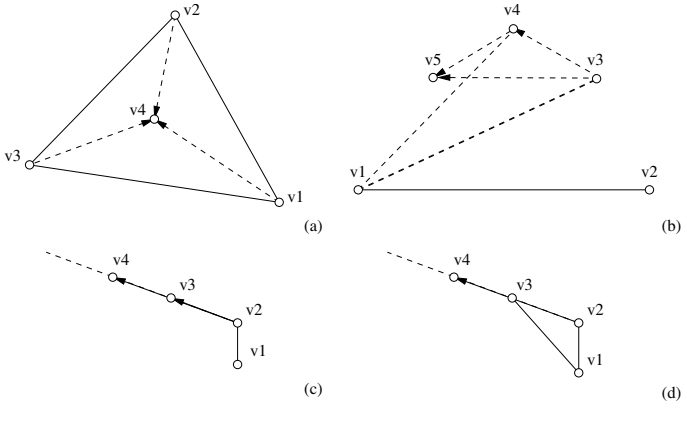

Fig. 6. Properties of left orientation primitive

two theorems (Figure 6 (c) and (d)) are used in [15] to handle degenerate cases. Actually, they are not needed in our approach, since LTURN_TRIAN already covers these cases. This illustrates the advantages of our approach: We always address general and degenerate cases at the same time, which makes the description succinct and readable. The same holds for later implementations that are made with three-valued data types.

4 Proof Techniques

In the previous section, we presented a way to specify geometric properties and algorithms with the help of three-valued logic. If a geometric algorithm is verified and all decisions of the algorithm depend on three-valued primitives, it can be analysed systematically. The following section presents theorems, conversions, and tactics to simplify this task.

4.1 Three-Valued Logic

Ternary Algebra. The system $\langle \mathbb{T}, \ddot{\vee}, \ddot{\wedge}, \ddot{\neg}, \mathsf{F}, \mathsf{T}, \mathsf{U} \rangle$ is a ternary algebra [3]: 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:

CONJ_TERNARY $\vdash\ a \ddot{\wedge} \ddot{\neg} a \ddot{\wedge} \mathsf{U} = a \ddot{\wedge} \ddot{\neg} a$
DISJ_TERNARY $\vdash\ a \ddot{\vee} \ddot{\neg} a \ddot{\vee} \mathsf{U} = a \ddot{\vee} \ddot{\neg} a$

Variations of Two-Valued Tactics. 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_EQ_TAC reduces a goal of equivalence of three-valued terms to forward and backward implication. When applied to a goal $A \vdash^? t_1 \ddot{\leftrightarrow} t_2$, the 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 of the theory.

Reduction to Two-Valued Terms. 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 subproposition 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:

  LOG3_CASES_EQ $\vdash (a = \mathsf{F}) \wedge (b = \mathsf{F}) \vee (a = \mathsf{U}) \wedge (b = \mathsf{U}) \vee$
  $(a = \mathsf{T}) \wedge (b = \mathsf{T}) = (a = b)$

  LOG3_CASES_LEQ $\vdash (a = \mathsf{F}) \wedge (b = \mathsf{F}) \vee a \leq \mathsf{U} \wedge (b = \mathsf{U}) \vee (b = \mathsf{T}) = a \leq b$

  LOG3_CASES_GEQ $\vdash (b = \mathsf{F}) \vee a \geq \mathsf{U} \wedge (b = \mathsf{U}) \vee (a = \mathsf{T}) \wedge (b = \mathsf{T}) = a \geq b$

  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 proposition 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 $P = \mathsf{U}$.

  LOG3_LEQ_GEQ_UU $\vdash a \leq \mathsf{U} \wedge a \geq \mathsf{U} = (a = \mathsf{U})$
- Elimination of depending 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:

  LOG3_NOT_CALC $\vdash ((\ddot{\neg} t = \mathsf{F}) = (t = \mathsf{T})) \wedge ((\ddot{\neg} t = \mathsf{T}) = (t = \mathsf{F})) \wedge$
  $(\ddot{\neg} t \leq \mathsf{U} = t \geq \mathsf{U}) \wedge (\ddot{\neg} t \geq \mathsf{U} = t \leq \mathsf{U})$

  LOG3_AND_CALC $\vdash (a \mathbin{\ddot{\wedge}} b = \mathsf{F}) = (a = \mathsf{F}) \vee (b = \mathsf{F})) \wedge$
  $(a \mathbin{\ddot{\wedge}} b = \mathsf{T}) = (a = \mathsf{T}) \wedge (b = \mathsf{T})) \wedge$
  $(a \mathbin{\ddot{\wedge}} b) \leq \mathsf{U} = a \leq \mathsf{U} \vee b \leq \mathsf{U}) \wedge$
  $(a \mathbin{\ddot{\wedge}} b) \geq \mathsf{U} = a \geq \mathsf{U} \wedge b \geq \mathsf{U}$

  LOG3_EXT_CALC $\vdash ((\ddot{\Delta}\, a = \mathsf{F}) = \neg a) \wedge ((\ddot{\Delta}\, a = \mathsf{T}) = a) \wedge$
  $(\ddot{\Delta}\, a \leq \mathsf{U} = \neg a) \wedge \ddot{\Delta}\, a \geq \mathsf{U} = a$

  LOG3_FORALL_CALC $\vdash ((\ddot{\forall} x.\, P(x) = \mathsf{F}) = \exists b.\, P(b) = \mathsf{F}) \wedge$
  $((\ddot{\forall} x. P(x) = \mathsf{T}) = \forall b.\, P(b) = \mathsf{T}) \wedge$
  $((\ddot{\forall} x. P(x) \leq \mathsf{U}) = \exists b.\, P(b) \leq \mathsf{U}) \wedge$
  $(\ddot{\forall} x. P(x) \geq \mathsf{U}) = \forall b.\, P(b) \geq \mathsf{U}$
- Elimination of negative terms: All two-valued negations in front of subterms can be eliminated, leaving better understandable expressions.

  LOG3_NOT2_CALC $\vdash (\neg(a = \mathsf{F}) = a \geq \mathsf{U}) \wedge (\neg(a = \mathsf{T}) = a \leq \mathsf{U}) \wedge$
  $(\neg(a = \mathsf{U}) = (a = \mathsf{F}) \vee (a = \mathsf{T})) \wedge$
  $(\neg(a \leq \mathsf{U}) = (a = \mathsf{T})) \wedge \neg(a \geq \mathsf{U}) = (a = \mathsf{F})$

  LOG3_ABS_NOT $\vdash (\ddot{\Delta}\, \neg a) = \ddot{\neg}(\ddot{\Delta}\, a)$

4.2 Vectors and Rational Numbers

Conversions and tactics that calculate vector and rational number expressions are provided. `VEC_CALCTERM_TAC` applies the calculation rules to a term, `VEC_CALC_TAC` to all terms of the type $\mathbb{Q}^2$. With the help of these tactics and the two theorems `VEC_EQ` and `RAT_EQ`, the equality of two vectors is reduced to the equalities between integers, which can be solved by the integer decision procedures of the HOL system. In this way, a lot of simple theorems can be automatically proven.

4.3 Example

We illustrate our approach by the convex hull algorithm presented in [6]. It divides the computation of the convex hull into two parts: the upper part and the lower part of the hull (see Figure 7 (a)). In this section, we focus on the construction of the lower part.

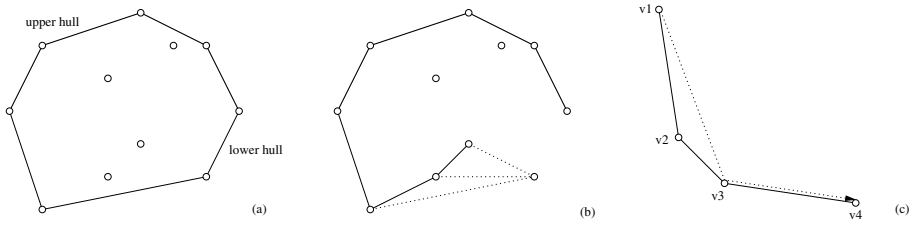

Fig. 7. Computation of the convex hull

Formalisation. The algorithm takes a list of points $\mathcal{L}$, which is sorted in lexicographic order (denoted as $\mathsf{lexSorted}(\mathcal{L})$), i.e. points are first sorted by their x-coordinates and if the x-coordinates should be the same, then the y-coordinates determines the ordering. The points are iteratively added to the lower part of the convex hull. After each addition, it is checked whether the last three points make 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 7 (b) illustrates this procedure. Formally, the construction of the lower hull can be described by the following functions[1]:

$$
\begin{aligned}
\texttt{normalise\_lower} \vdash_{\text{def}} \; & (\mathsf{normLow}([\,]) = [\,]) \wedge \\
& (\mathsf{normLow}([e_1]) = [e_1]) \wedge \\
& (\mathsf{normLow}([e_1; e_2]) = [e_1; e_2]) \wedge \\
& (\mathsf{normLow}((e_1 :: e_2 :: e_3 :: \mathcal{L})) = \\
& \qquad \mathbf{if}\ \mathsf{lturn}(e_1, e_2, e_3) = \mathsf{T}\ \mathbf{then}\ e_1 :: e_2 :: e_3 :: \mathcal{L} \\
& \qquad \mathbf{else}\ \mathsf{normLow}((e_1 :: e_3 :: \mathcal{L})))
\end{aligned}
$$

$$
\begin{aligned}
\texttt{hull\_lower} \vdash_{\text{def}} \; & (\mathsf{hullLow}([\,]) = [\,]) \wedge \\
& (\mathsf{hullLow}(e :: \mathcal{L}) = \mathsf{normLow}(e :: \mathsf{hullLow}(\mathcal{L})))
\end{aligned}
$$

[1] $[\,]$ denotes the empty list, $[e_1; e_2]$ 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 $\mathcal{L}$.

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 $\mathsf{hullLow}$ 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 passing those points. We define the corresponding predicate $\mathsf{lconvex}$ 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 (lpoint), and all points must lie on the left side of the edge that is created by the insertion of the new point (ledge).

$$
\begin{aligned}
\texttt{left\_edge} \vdash_{\text{def}}\ & (\mathsf{ledge}(e_1, e_2, [\,])) \wedge \\
& (\mathsf{ledge}(e_1, e_2, e :: \mathcal{L}) = (\mathsf{lturn}(e, e_1, e_2) = \mathsf{T}) \wedge \mathsf{ledge}(e_1, e_2, \mathcal{L})) \\
\texttt{left\_point} \vdash_{\text{def}}\ & (\mathsf{lpoint}(e, [\,])) \wedge \\
& (\mathsf{lpoint}(e, [e_1])) \wedge \\
& (\mathsf{lpoint}(e, e_1 :: e_2 :: t) = \\
& \quad (\mathsf{lturn}(e, e_2, e_1) = \mathsf{T}) \wedge \mathsf{lpoint}(e, e_2 :: \mathcal{L})) \\
\texttt{left\_convex} \vdash_{\text{def}}\ & (\mathsf{lconvex}([\,])) \wedge \\
& (\mathsf{lconvex}([e_1])) \wedge \\
& (\mathsf{lconvex}(e_1 :: e_2 :: \mathcal{L}) = \\
& \quad \mathsf{ledge}(e_1, e_2, \mathcal{L}) \wedge \mathsf{lpoint}(e_1, e_2 :: \mathcal{L}) \wedge \mathsf{lconvex}(e_2 :: \mathcal{L}))
\end{aligned}
$$

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 make a left turn.

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

Then, under the condition of a lexicographic ordering a kind of transitivity (see Figure 7 (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 results the desired theorem `CVX_LOWER`.

$$
\begin{aligned}
\texttt{CVX\_TRANS\_LOWER} \vdash\ & (\mathsf{lturn}(v_1, v_2, v_3) = \mathsf{T}) \wedge (\mathsf{lturn}(v_2, v_3, v_4) = \mathsf{T}) \wedge \\
& (v_1 \prec_{\mathsf{lex}} v_2 = \mathsf{T}) \wedge (v_2 \prec_{\mathsf{lex}} v_3 = \mathsf{T}) \wedge (v_3 \prec_{\mathsf{lex}} v_4 = \mathsf{T}) \\
& \Rightarrow (\mathsf{lturn}(v_1, v_3, v_4) = \mathsf{T}) \\
\texttt{CVX\_LOWER} \vdash\ & \mathsf{lexSorted}(\mathcal{L}) \wedge \mathsf{lchain}(\mathcal{L}) \Rightarrow \mathsf{lconvex}(\mathcal{L})
\end{aligned}
$$

Note that in the proofs, we do not have to address the degenerate cases explicitely. We exploit 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.

5 Conclusions

In this paper, we addressed the problem of specifying and verifying algorithms of computational geometry. Starting from applications like motion planning or collision detection, we formalised basic geometric objects and primitives used in analytic geometry. The main contribution of this paper is to consistently use three-valued logic for this purpose. To this end, we defined a three-valued logic in the theorem prover HOL and used it for the formalisation of geometric primitives in the presence of degenerate cases. Using the HOL theorem prover, we proved numerous theorems and provided various tactics for automating parts of proofs. In particular, we use efficient tactics to translate three-valued goals to two-valued ones. In this way, conventional tactics and proof tools can be used for automated reasoning.

We evaluated our approach by small examples. They all show that our approach is very suitable: The specifications are both precise and compact; the integrated consideration of degenerate cases with the help of three-valued logic makes both algorithms and proofs simpler and clearer. At the same time, all advantages of traditional proof techniques are preserved due to the possible reduction to two-valued expressions.

Our next and more ambitious verification project is the development of a formally proven map overlay algorithm [6] that is suited for applications in safety-critical embedded systems. For this algorithm, some more foundations are required, as e.g. an appropriate formalisation of plane graphs [19].

References

1. L. Bolc and P. Borowik. *Many-Valued logics*. Springer, 1992.
2. J. Brandt and K. Schneider. Dependable polygon-processing algorithms for safety-critical embedded systems. In *International Conference on Embedded And Ubiquitous Computing (EUC)*, LNCS, Nagasaki, Japan, 2005. Springer.
3. J. Brzozowski and C.-J. Seger. *Asynchronous Circuits*. Springer, 1995.
4. 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.
5. S. Chou, X. Gao, and J. Zhang. *Machine Proofs in Geometry*. World Scientific, Singapore, 1994.
6. M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry*. Springer, 2000.
7. H. Edelsbrunner and E. Mücke. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. *ACM Transactions on Graphics*, 9(1):66–104, 1990.
8. E. Eichelberger. Hazard detection in combinational and sequential switching circuits. *IBM Journal of Research and Development*, 9:90–99, 1965.
9. J. Foley, A. van Dam, S. Feiner, and J. Hughes. *Computer Graphics: Principles and Practice*. Addison Wesley, 2000.
10. M. Gordon and T. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.
11. S. Kleene. *Introduction to Metamathematics*. North Holland, 1952.
12. D. Knuth. *Axioms and Hulls*, volume 606 of *LNCS*. Springer, 1992.
13. S. Malik. Analysis of cycle combinational circuits. *IEEE Transactions on Computer Aided Design*, 13(7):950–956, July 1994.

14. K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.
15. D. Pichardie and Y. Bertot. Formalizing convex hull algorithms. In R. Boulton and P. Jackson, editors, *Higher Order Logic Theorem Proving and its Applications (TPHOL)*, volume 2152 of *LNCS*, pages 346–361, Edinburgh, Scotland, UK, 2001. Springer.
16. T. Reps, M. Sagiv, and R. Wilhelm. Static program analysis via 3-valued logic. In R. Alur and D. Peled, editors, *Conference on Computer Aided Verification (CAV)*, volume 3114 of *LNCS*, pages 15–30, Boston, MA, USA, 2004. Springer.
17. T. Schuele and K. Schneider. Three-valued logic in bounded model checking. In *Formal Methods and Models for Codesign (MEMOCODE)*, Verona, Italy, 2005. IEEE Computer Society.
18. W.-T. Wu. On the decision problem and the mechanization of theorem proving in elementary geometry. *Scientia Sinica*, 21:157–179, 1978.
19. M. Yamamoto, S. Nishizaki, and M. Hagiya. Formalization of planar graphs. In E. Schubert, P. Windley, and J. Alves-Foss, editors, *Higher Order Logic Theorem Proving and its Applications (TPHOL)*, volume 971 of *LNCS*, pages 369–384, Aspen Grove, Utah, USA, September 1995. Springer.